Advanced Data Structures and Algorithm Analysis

丁尧相
浙江大学

Fall and Winter 2025
Lecture 10

Approximation Algorithms

- Bin packing
- 0-1 knapsack
- K-center selection
- Take-home messages

Approximation Algorithms

- Bin packing
- 0-1 knapsack
- K-center selection
- Take-home messages

Coping with NP-completeness

Q. Suppose I need to solve an **NP**-hard optimization problem.
What should I do?

Coping with NP-completeness

Q. Suppose I need to solve an **NP**-hard optimization problem.
What should I do?

A. Sacrifice one of three desired features.

i. Runs in polynomial time.
ii. Solves arbitrary instances of the problem.
iii. Finds optimal solution to problem.

Coping with NP-completeness

Q. Suppose I need to solve an **NP**-hard optimization problem.
What should I do?

A. Sacrifice one of three desired features.

i. Runs in polynomial time.
ii. Solves arbitrary instances of the problem.
iii. Finds optimal solution to problem.

ρ-approximation algorithm.

- Runs in polynomial time.
- Solves arbitrary instances of the problem
- Finds solution that is within ratio ρ of optimum.

Challenge. Need to prove a solution's value is close to optimum, without even knowing what is optimum value.

Approximate Bin Packing

Approximate Bin Packing

Given N items of sizes $S_1 , S_2 , \ldots, S_N$, such that $0 < S_i \leq 1$ for all $1 \leq i \leq N$. Pack these items in the fewest number of bins, each of which has unit capacity.

Approximate Bin Packing

Given N items of sizes S_1 , S_2 , …, S_N , such that $0 < S_i \le 1$ for all $1 \le i \le N$. Pack these items in the **fewest** number of bins, each of which has **unit capacity**.

〖Example〗 $N = 7$; $S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8$

Approximate Bin Packing

Given N items of sizes S_1 , S_2 , …, S_N , such that $0 < S_i \leq 1$ for all $1 \leq i \leq N$. Pack these items in the fewest number of bins, each of which has unit capacity.

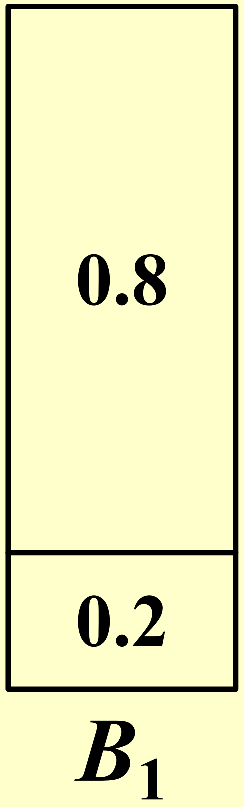
〖Example〗 $N = 7$; $S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8$

An Optimal Packing

Approximate Bin Packing

Given N items of sizes $S_1, S_2, \ldots, S_N$, such that $0 < S_i \leq 1$ for all $1 \leq i \leq N$. Pack these items in the fewest number of bins, each of which has unit capacity.

〖Example〗 $N = 7$; $S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8$

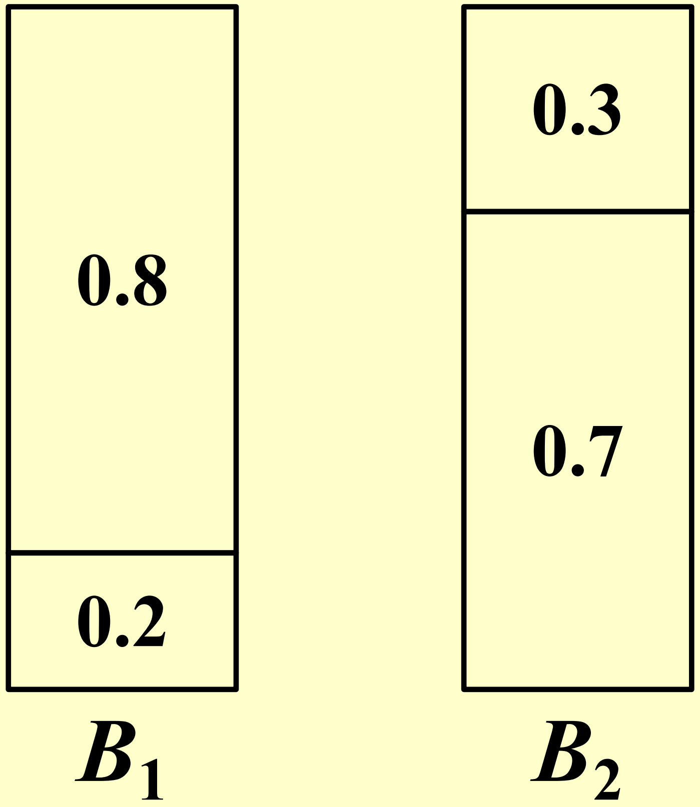


An Optimal Packing

Approximate Bin Packing

Given N items of sizes $S_1, S_2, \ldots, S_N$, such that $0 < S_i \le 1$ for all $1 \le i \le N$. Pack these items in the fewest number of bins, each of which has unit capacity.

〖Example〗 $N = 7$; $S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8$

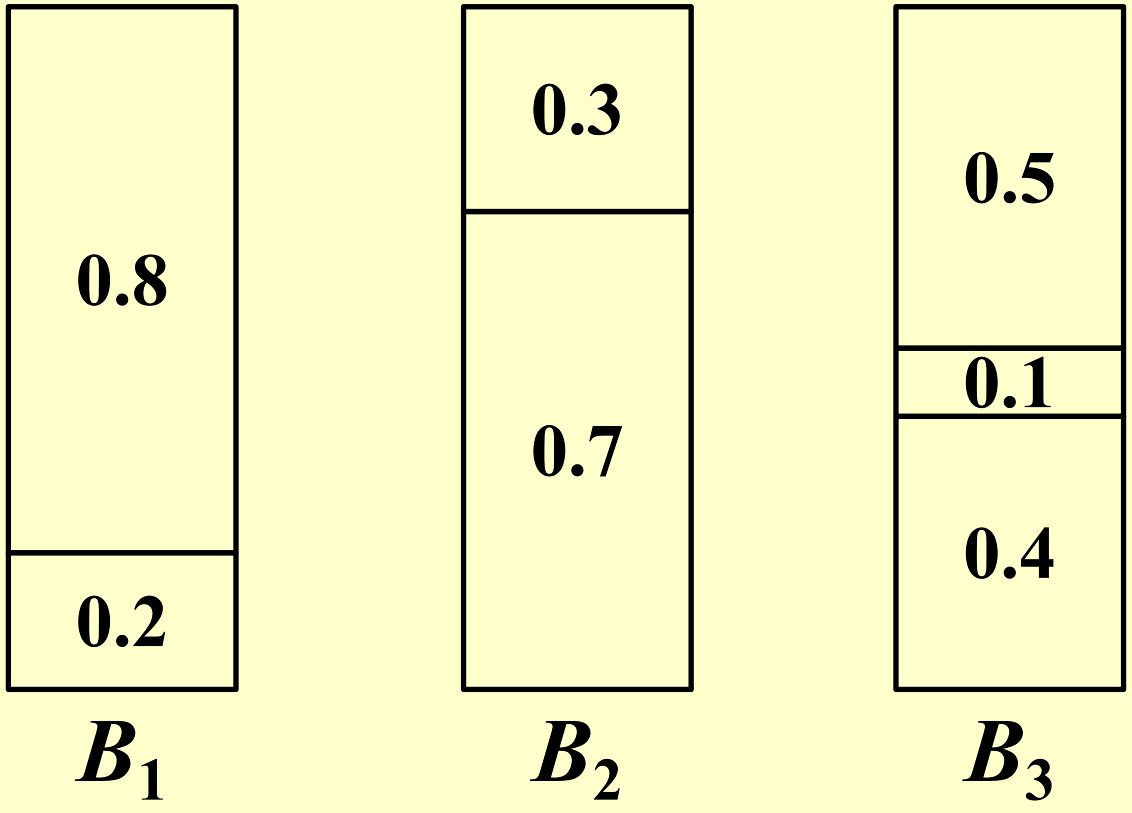


An Optimal Packing

Approximate Bin Packing

Given N items of sizes $S_1, S_2, \ldots, S_N$, such that $0 < S_i \le 1$ for all $1 \le i \le N$. Pack these items in the fewest number of bins, each of which has unit capacity.

【Example】 $N = 7$; $S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8$

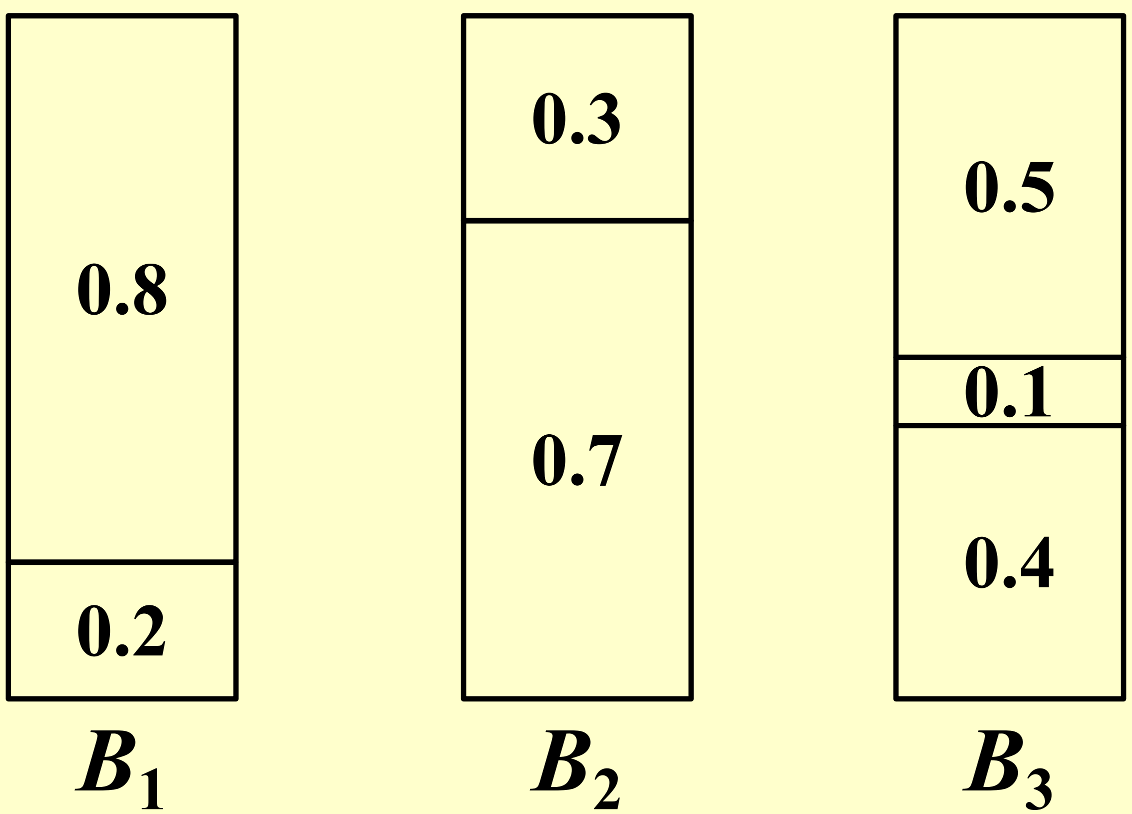


An Optimal Packing

Approximate Bin Packing

Given N items of sizes $S_1, S_2, \ldots, S_N$, such that $0 < S_i \leq 1$ for all $1 \leq i \leq N$. Pack these items in the fewest number of bins, each of which has unit capacity.

〖Example〗 $N = 7$; $S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8$

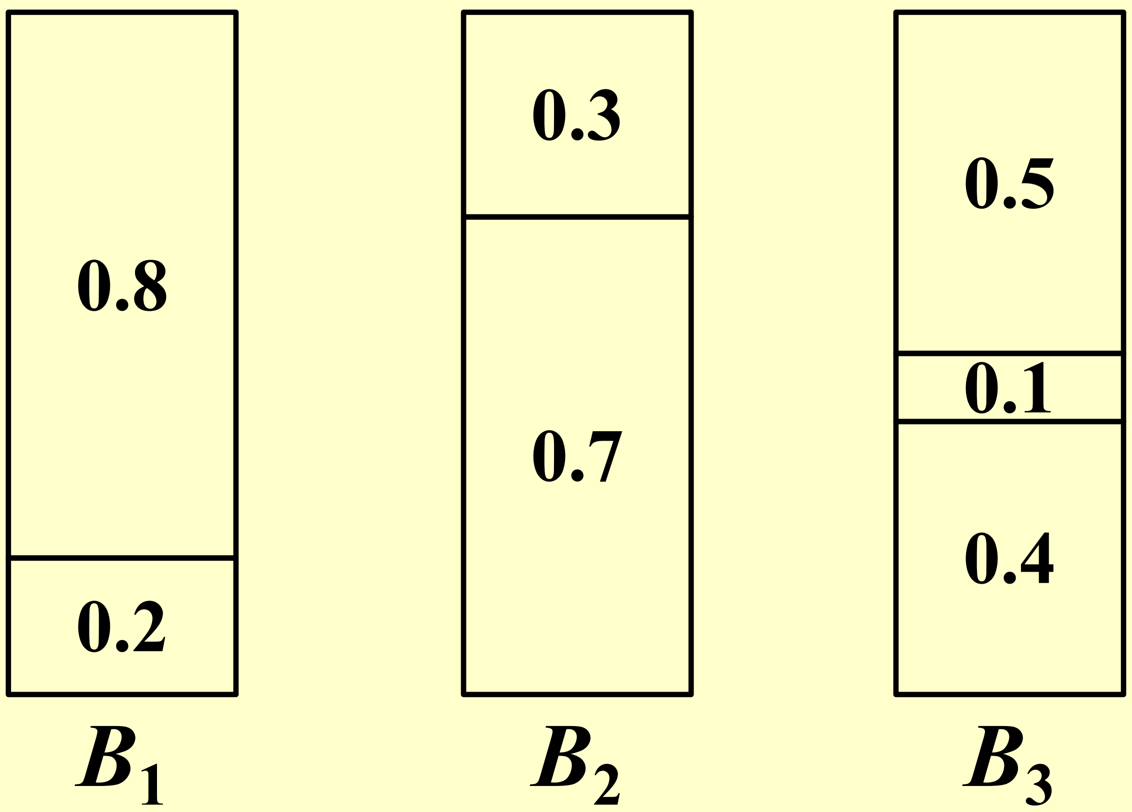


An Optimal Packing

Approximate Bin Packing

Given N items of sizes $S_1, S_2, \ldots, S_N$, such that $0 < S_i \le 1$ for all $1 \le i \le N$. Pack these items in the **fewest** number of bins, each of which has **unit capacity**.

【Example】 $N = 7$; $S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8$



An Optimal Packing

NP Hard

The NP-complete 2-partition problem can be reduced to bin packing.
See Slides Page 13.

- **Next Fit**

❖ Next Fit

```
void NextFit ( )
{   read item1;
    while ( read item2 ) {
        if ( item2 can be packed in the same bin as item1 )
              place item2 in the bin;
        else
              create a new bin for item2;
        item1 = item2;
    } /* end-while */
}
```

❖ Next Fit

```
void NextFit ( )
{   read item1;
    while ( read item2 ) {
       if ( item2 can be packed in the same bin as item1 )
             place item2 in the bin;
       else
             create a new bin for item2;
       item1 = item2;
    } /* end-while */
}
```

【Theorem】 Let M be the optimal number of bins required to pack a list I of items. Then *next fit* never uses more than $2M - 1$ bins. There exist sequences such that *next fit* uses $2M - 1$ bins.

A simple proof for Next Fit:

A simple proof for Next Fit:

If Next Fit generates $2M$ (or $2M+1$) bins, then the optimal solution must generate at least $M+1$ bins.

A simple proof for Next Fit:

If Next Fit generates $2M$ (or $2M+1$) bins, then the optimal solution must generate at least $M+1$ bins.

Let $S(B_i)$ be the size of the ith bin. Then we must have:

A simple proof for Next Fit:

If Next Fit generates $2M$ (or $2M+1$) bins, then the optimal solution must generate at least $M+1$ bins.

Let $S(B_i)$ be the size of the ith bin. Then we must have:

$S(B_1) + S(B_2) > 1$

$S(B_3) + S(B_4) > 1$

…………

$S(B_{2M-1}) + S(B_{2M}) > 1$

A simple proof for Next Fit:

If Next Fit generates $2M$ (or $2M+1$) bins, then the optimal solution must generate at least $M+1$ bins.

Let $S(B_i)$ be the size of the ith bin. Then we must have:

$S(B_1) + S(B_2) > 1$

$S(B_3) + S(B_4) > 1$

…………

$S(B_{2M-1}) + S(B_{2M}) > 1$

$$\Rightarrow \quad \sum_{i=1}^{2M} S(B_i) > M$$

A simple proof for Next Fit:

If Next Fit generates $2M$ (or $2M+1$) bins, then the optimal solution must generate at least $M+1$ bins.

Let $S(B_i)$ be the size of the ith bin. Then we must have:

$S(B_1) + S(B_2) > 1$

$S(B_3) + S(B_4) > 1$

…………

$S(B_{2M-1}) + S(B_{2M}) > 1$

$$\Rightarrow \quad \sum_{i=1}^{2M} S(B_i) > M$$

The optimal solution needs at least $\lceil$total size of all the items$\rceil$ bins

A simple proof for Next Fit:

If Next Fit generates $2M$ (or $2M+1$) bins, then the optimal solution must generate at least $M+1$ bins.

Let $S(B_i)$ be the size of the ith bin. Then we must have:

$S(B_1) + S(B_2) > 1$

$S(B_3) + S(B_4) > 1$

…………

$S(B_{2M-1}) + S(B_{2M}) > 1$

$$\Rightarrow \quad \sum_{i=1}^{2M} S(B_i) > M$$

The optimal solution needs at least $\lceil$total size of all the items$\rceil$ bins

$$\Rightarrow \quad \lceil \text{total size of all the items} \rceil = \left\lceil \sum_{i=1}^{2M} S(B_i) \right\rceil \geq M + 1$$

❖ First Fit

❖ First Fit

```
void FirstFit ( )
{   while ( read item ) {
        scan for the first bin that is large enough for item;
        if ( found )
            place item in that bin;
        else
            create a new bin for item;
    } /* end-while */
}
```

❖ First Fit

```
void FirstFit ( )
{   while ( read item ) {
        scan for the first bin that is large enough for item;
        if ( found )
            place item in that bin;
        else
            create a new bin for item;
    } /* end-while */
}
```

Can be implemented in O(N log N)

❖ First Fit

```
void FirstFit ( )
{   while ( read item ) {
        scan for the first bin that is large enough for item;
        if ( found )
            place item in that bin;
        else
            create a new bin for item;
    } /* end-while */
}
```

Can be implemented in $O(N \log N)$

【Theorem】 Let M be the optimal number of bins required to pack a list I of items. Then *first fit* never uses more than $17M / 10$ bins. There exist sequences such that *first fit* uses $17(M - 1) / 10$ bins.

❖ First Fit

```
void FirstFit ( )
{   while ( read item ) {
        scan for the first bin that is large enough for item;
        if ( found )
            place item in that bin;
        else
            create a new bin for item;
    } /* end-while */
}
```

Can be implemented in O(N log N)

【Theorem】Let M be the optimal number of bins required to pack a list I of items. Then *first fit* never uses more than $17M / 10$ bins. There exist sequences such that *first fit* uses $17(M – 1) / 10$ bins.

❖ Best Fit

Place a new item in the tightest spot among all bins.

❖ First Fit

```
void FirstFit ( )
{   while ( read item ) {
        scan for the first bin that is large enough for item;
        if ( found )
            place item in that bin;
        else
            create a new bin for item;
    } /* end-while */
}
```

Can be implemented in O(N log N)

【Theorem】 Let M be the optimal number of bins required to pack a list I of items. Then *first fit* never uses more than $17M / 10$ bins. There exist sequences such that *first fit* uses $17(M – 1) / 10$ bins.

❖ Best Fit

Place a new item in the tightest spot among all bins.

T = O(N log N) and bin no. $\leq 1.7M$

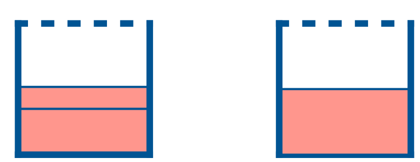
SOL(First Fit) $\leq$ 2OPT

Instance: n items with sizes $s_1, s_2, \ldots, s_n \in (0, 1]$;
Pack them into *smallest* number of *unit-sized* bins.

FirstFit

Initially $k=1$;
for $i=1,2, \ldots, n$
  item i joins the *first* bin among $1,2, \ldots, k$ in which it *fits*;
  if item i can fit into none of these k bins
    open a new bin k++ and item i joins it;

Observation: All but at most one bin are more than half full.



$\Sigma_i\, s_i > (SOL - 1) / 2$ $\quad$ $SOL - 1 < 2\, \Sigma_i\, s_i \leq 2\, OPT$

$OPT \geq \Sigma_i\, s_i$ $\quad$ $SOL \leq 2\, OPT$

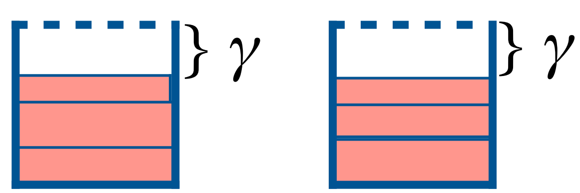
Instance: n items with sizes $s_1, s_2, \ldots, s_n \in (0, 1]$;
Pack them into *smallest* number of *unit-sized* bins.

FirstFit

Initially $k=1$;
for $i=1,2, \ldots, n$
  item i joins the *first* bin among $1,2, \ldots, k$ in which it *fits*;
  if item i can fit into none of these k bins
    open a new bin k++ and item i joins it;

Assumption: If all items are small, $s_i < \gamma < 0.5$

Observation: All but at most one bin are more than $(1\text{-}\gamma)$ full.



$\sum_i s_i > (1\text{-}\gamma)(SOL - 1)$ $\Rightarrow$ $SOL \le OPT / (1\text{-}\gamma) + 1$

〖Example〗 $S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8$

〖Example〗 $S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8$

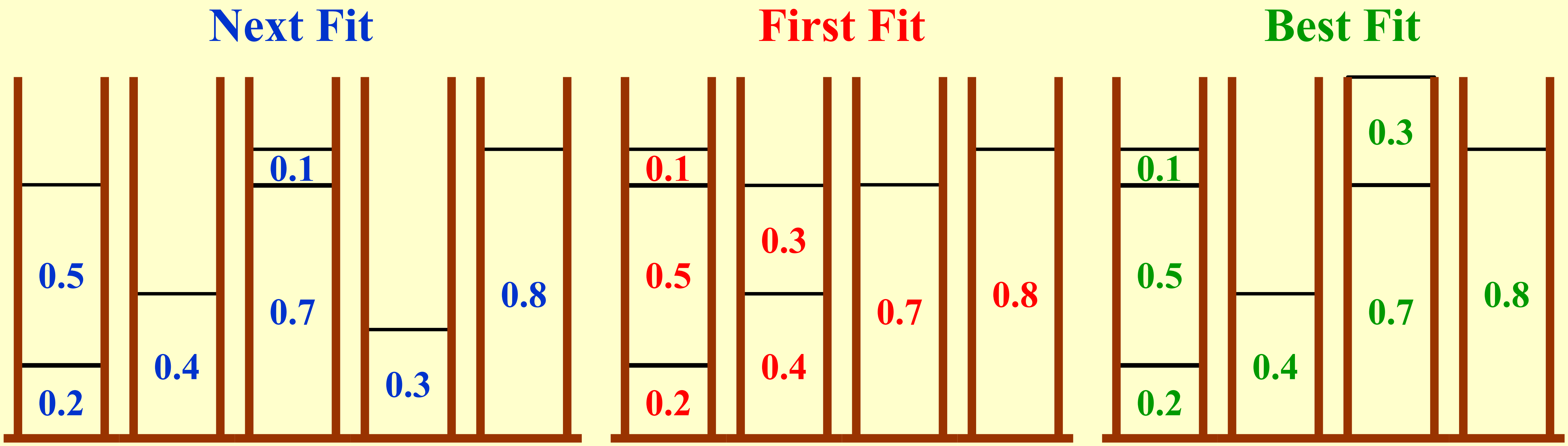
Next Fit **First Fit** **Best Fit**

〖Example〗 $S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8$

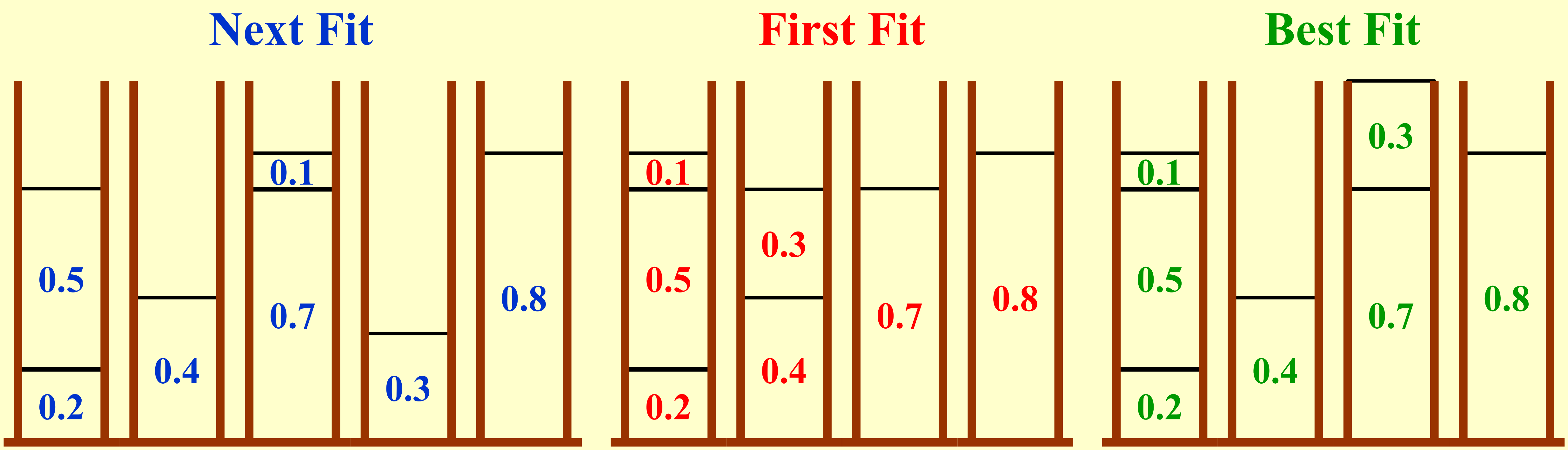
Next Fit | First Fit | Best Fit

Discussion 14: Please show the results.

〖Example〗 S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8



〖Example〗 S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8



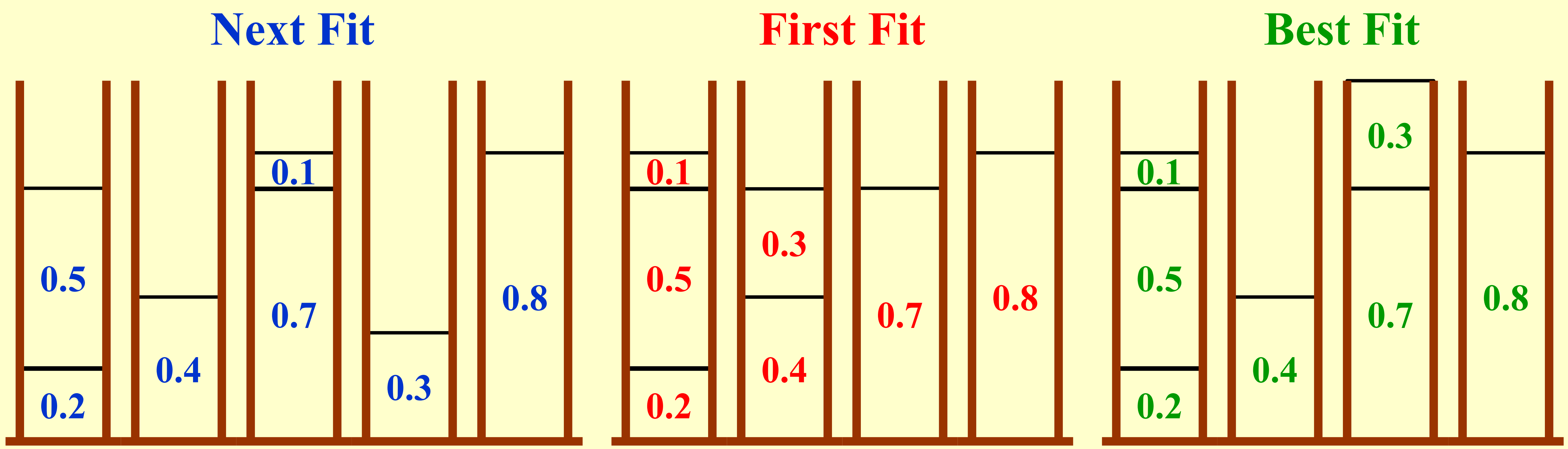
〖Example〗 S_i = 1/7+ε, 1/7+ε, 1/7+ε, 1/7+ε, 1/7+ε, 1/7+ε,
1/3+ε, 1/3+ε, 1/3+ε, 1/3+ε, 1/3+ε, 1/3+ε,
1/2+ε, 1/2+ε, 1/2+ε, 1/2+ε, 1/2+ε, 1/2+ε

where ε = 0.001.

☞ The optimal solution requires ? bins.

However, all the three on-line algorithms require ? bins.

〖Example〗 S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8



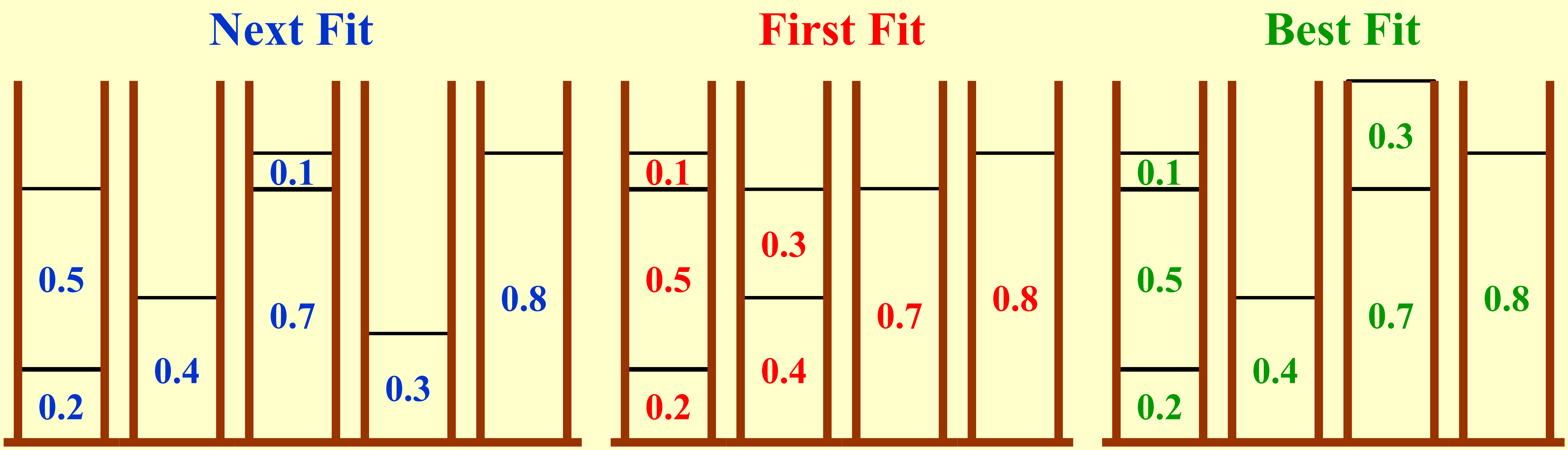
〖Example〗 S_i = 1/7+ε, 1/7+ε, 1/7+ε, 1/7+ε, 1/7+ε, 1/7+ε,
1/3+ε, 1/3+ε, 1/3+ε, 1/3+ε, 1/3+ε, 1/3+ε,
1/2+ε, 1/2+ε, 1/2+ε, 1/2+ε, 1/2+ε, 1/2+ε

where ε = 0.001.

☞ The optimal solution requires 6 bins.

However, all the three on-line algorithms require ? bins.

【Example】 S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8



【Example】 S_i = 1/7+ε, 1/7+ε, 1/7+ε, 1/7+ε, 1/7+ε, 1/7+ε,
1/3+ε, 1/3+ε, 1/3+ε, 1/3+ε, 1/3+ε, 1/3+ε,
1/2+ε, 1/2+ε, 1/2+ε, 1/2+ε, 1/2+ε, 1/2+ε

where ε = 0.001.

☞ The optimal solution requires 6 bins.

However, all the three on-line algorithms require 10 bins.

☞ **On-line Algorithms**

Place an item before processing the next one, and can NOT change decision.

☞ **On-line Algorithms**

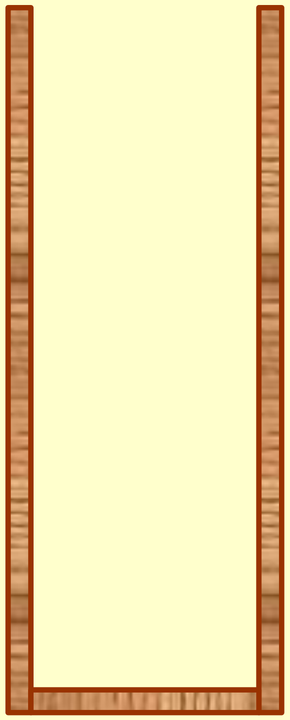
Place an item before processing the next one, and can NOT change decision.

〖Example〗 $S_i = 0.4$

☞ **On-line Algorithms**

Place an item before processing the next one, and can NOT change decision.

〖Example〗 $S_i = 0.4$



☞ On-line Algorithms

Place an item before processing the next one, and can NOT change decision.

〖Example〗 $S_i = 0.4$

☞ On-line Algorithms

Place an item before processing the next one, and can NOT change decision.

〖Example〗 $S_i = 0.4\ ,\ 0.4$

☞ **On-line Algorithms**

Place an item before processing the next one, and can NOT change decision.

〖Example〗 $S_i = 0.4 , 0.4$

☞ **On-line Algorithms**

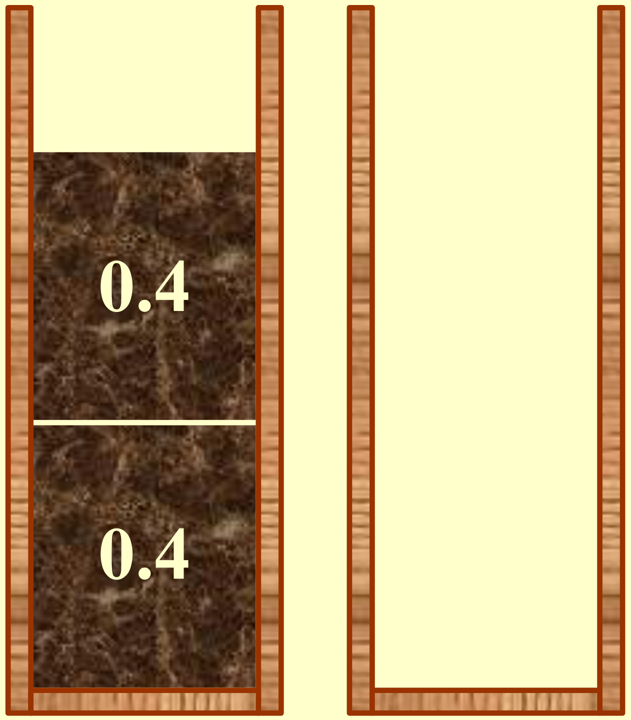
Place an item before processing the next one, and can NOT change decision.

〖Example〗 $S_i = 0.4\ , 0.4, 0.6$

☞ On-line Algorithms

Place an item before processing the next one, and can NOT change decision.

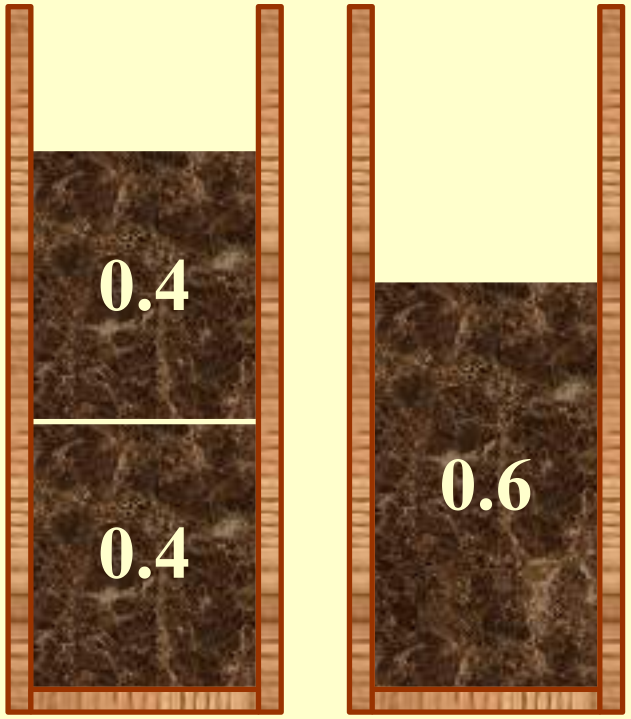
〖Example〗 $S_i = 0.4\ , 0.4, 0.6$



☞ **On-line Algorithms**

Place an item before processing the next one, and can NOT change decision.

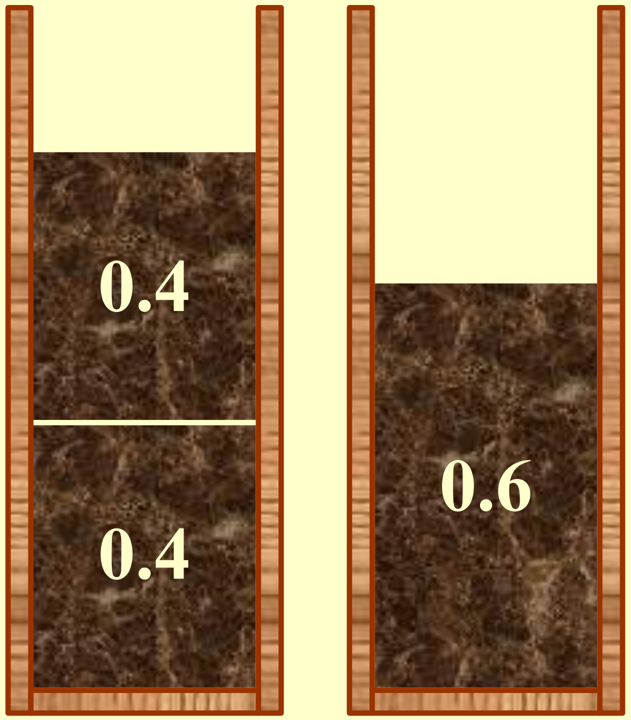
〖Example〗 $S_i = 0.4\ ,\ 0.4,\ 0.6$



☞ **On-line Algorithms**

Place an item before processing the next one, and can NOT change decision.

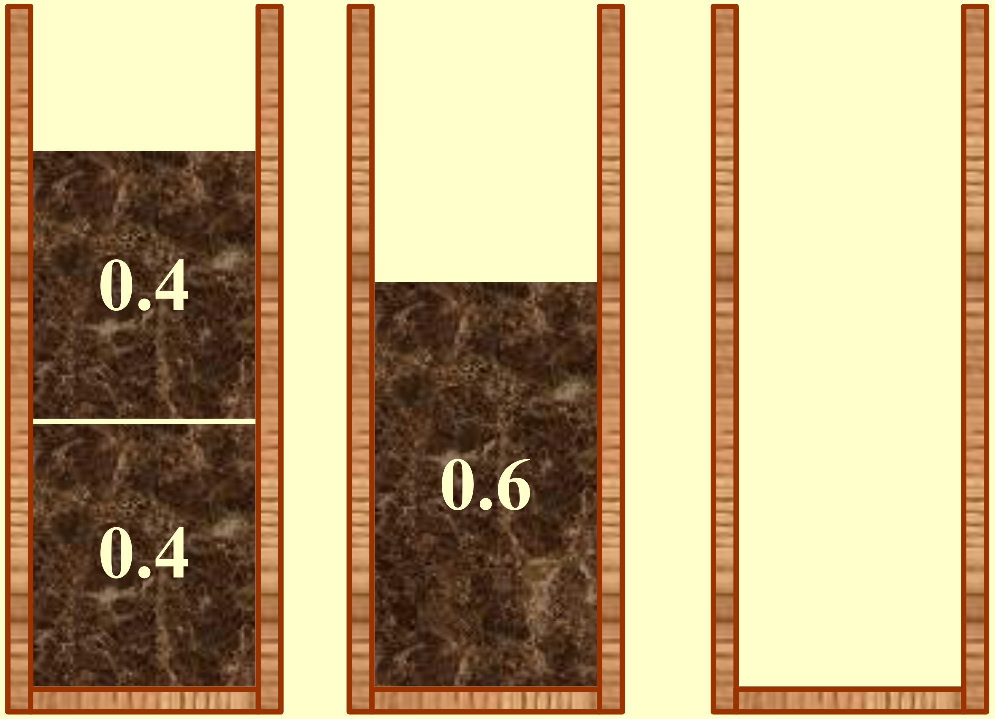
〖Example〗 S_i = 0.4 , 0.4, 0.6, 0.6



☞ On-line Algorithms

Place an item before processing the next one, and can **NOT change** decision.

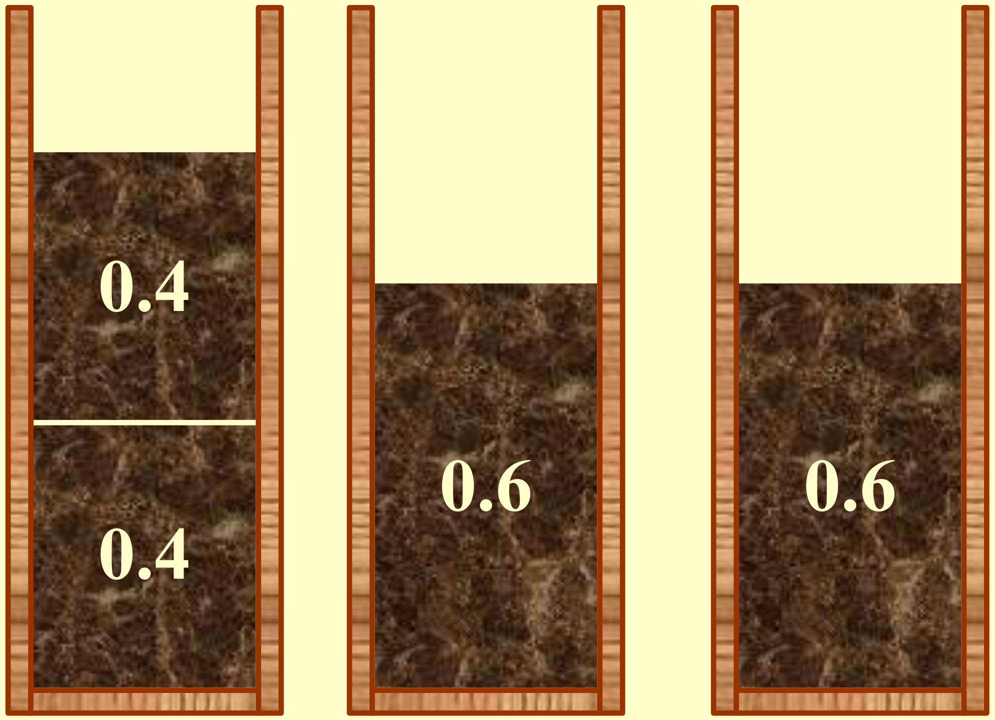
〖Example〗 S_i = 0.4 , 0.4, 0.6, 0.6



☞ **On-line Algorithms**

Place an item before processing the next one, and can NOT change decision.

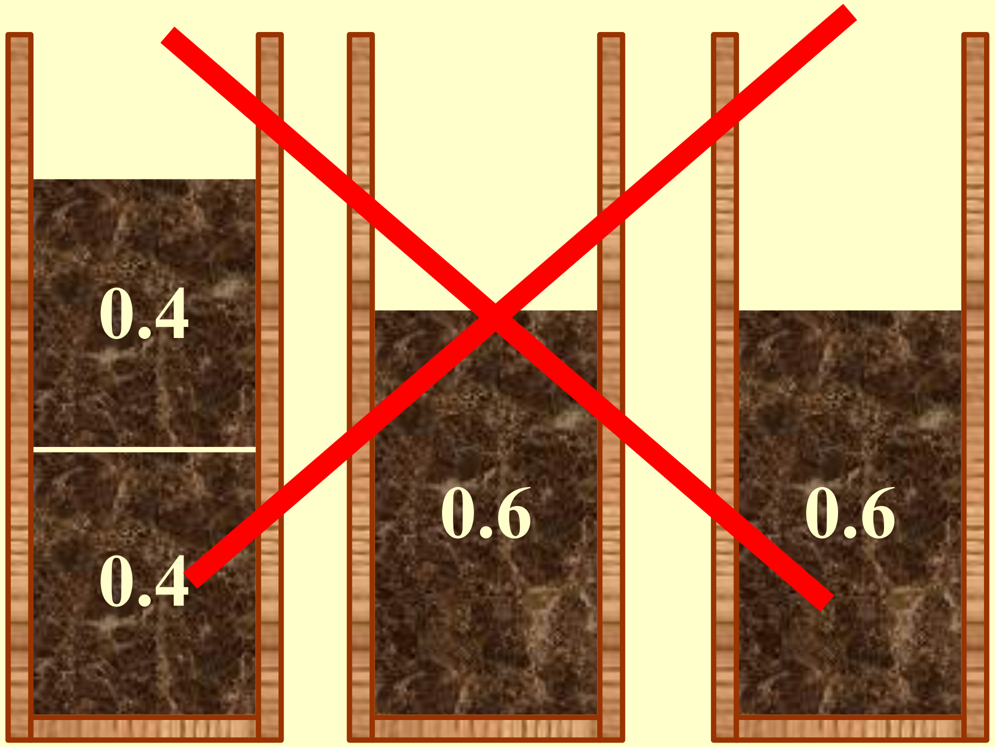
〖Example〗 $S_i = 0.4\ , 0.4, 0.6, 0.6$



☞ On-line Algorithms

Place an item before processing the next one, and can **NOT change** decision.

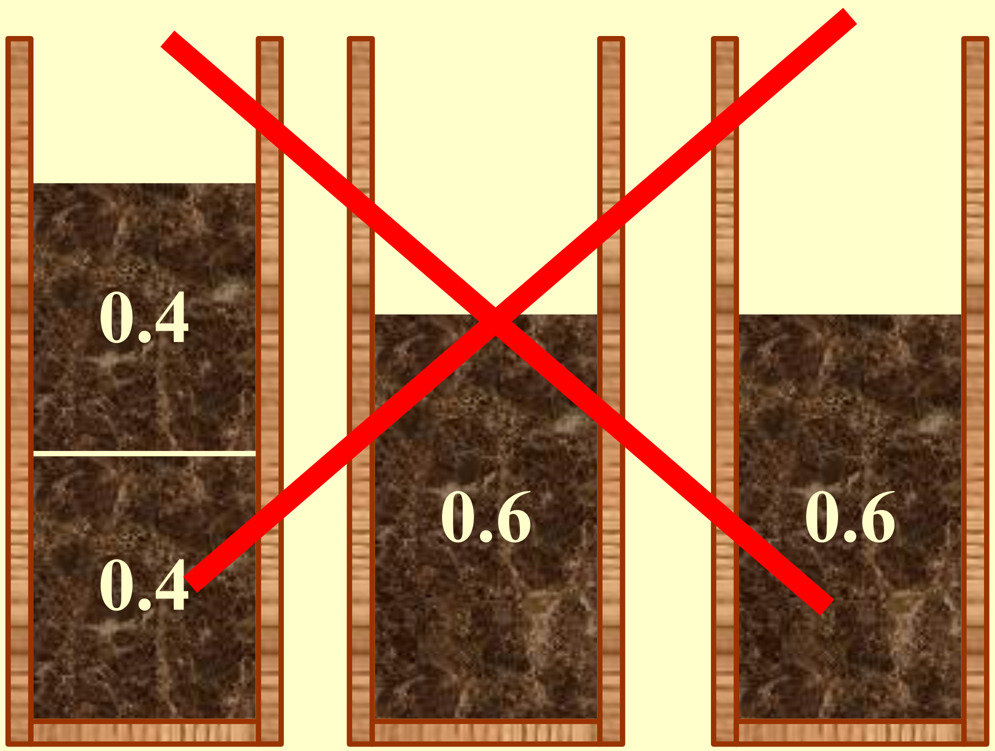
〖Example〗 $S_i = 0.4\ , 0.4, 0.6, 0.6$



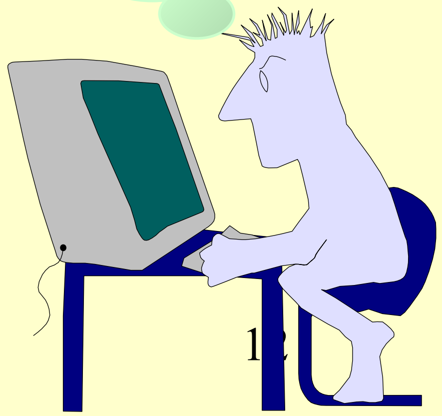
☞ On-line Algorithms

Place an item before processing the next one, and can **NOT change** decision.

〖Example〗 $S_i = 0.4\ , 0.4, 0.6, 0.6$



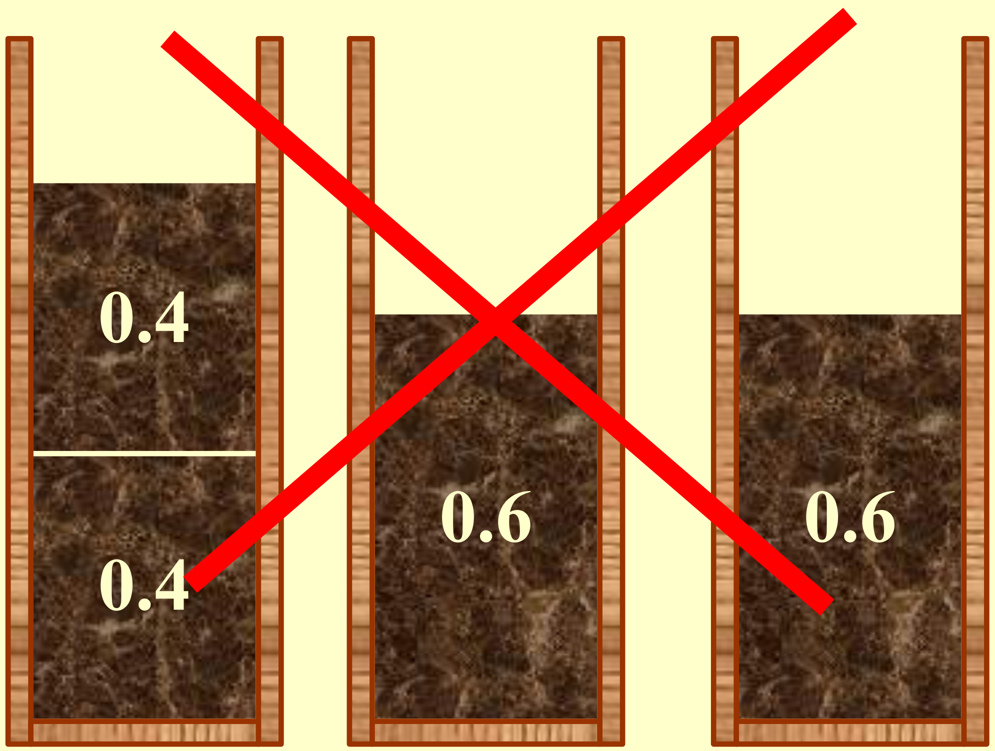
You never know
when the input might end.
No on-line algorithm
can always give
an optimal solution.



☞ On-line Algorithms

Place an item before processing the next one, and can **NOT change** decision.

〖Example〗 $S_i = 0.4\ , 0.4, 0.6, 0.6$



You never know
when the input might end.
No on-line algorithm
can always give
an optimal solution.

【Theorem】 There are inputs (in the previous slide) that force any on-line bin-packing algorithm to use at least 5/3 the optimal number of bins.

ELSEVIER

Journal of Computer and System Sciences

Volume 102, June 2019, Pages 1-17

The optimal absolute ratio for online bin packing ☆

János Balogh [a], József Békési [a], György Dósa [b], Jiří Sgall [c], Rob van Stee [d]

[a] Department of Applied Informatics, Gyula Juhász Faculty of Education, University of Szeged, H-6701 Szeged, POB 396, Hungary

[b] Department of Mathematics, University of Pannonia, H-8200 Veszprém, Hungary

[c] Computer Science Institute of Charles University, Faculty of Mathematics and Physics, Praha, Czech Republic

[d] Department of Mathematics, University of Siegen, Siegen, Germany

Lower Bound

Instance: n items with sizes $s_1, s_2, \ldots, s_n \in (0, 1]$;
Pack them into *smallest* number of *unit-sized* bins.

Theorem

Unless **P=NP**, there is no poly-time approximation algorithm for bin packing with approximation ratio $<3/2$.

reduction from the partition problem:

Input: n numbers $x_1, x_2, \ldots, x_n \in \mathbb{Z}^+$.

Determine whether $\exists$ a partition of $\{1, 2, \ldots, n\}$ into A and B such that $\sum_{i \in A} x_i = \sum_{i \in B} x_i$.

reduction n items with sizes $s_1, s_2, \ldots, s_n$ where $s_i = 2x_i / \sum_j x_j$

$\exists$ a packing into 2 unit-sized bins → "yes"

all packings use ≥ 3 unit-sized bins → "no"

partition problem

Instance: n items with sizes $s_1, s_2, \ldots, s_n \in (0, 1]$;
Pack them into *smallest* number of *unit-sized* bins.

Theorem

Unless **P=NP**, there is no poly-time approximation algorithm for bin packing with approximation ratio $<3/2$.

It is **NP**-hard to distinguish between:

- the instances with $\mathrm{OPT} = 2$;
- the instances with $\mathrm{OPT} \geq 3$.

FirstFitDecreasing (FFD)

Sort items in non-increasing order of sizes;
run **FirstFit**;

FFD returns a packing into $\leq 11/9\ OPT + 6/9$ bins

☞ **Off-line Algorithms**

View the entire item list before producing an answer.

☞ **Off-line Algorithms**

View the entire item list before producing an answer.

Trouble-maker: The large items

☞ **Off-line Algorithms**

View the entire item list before producing an answer.

Trouble-maker: The large items

Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.

☞ **Off-line Algorithms**

View the entire item list before producing an answer.

Trouble-maker: The large items

Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.

〖Example〗 S_i = **0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8**

☞ Off-line Algorithms

View the entire item list before producing an answer.

Trouble-maker: The large items

Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.

〖Example〗 S_i = **0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1**

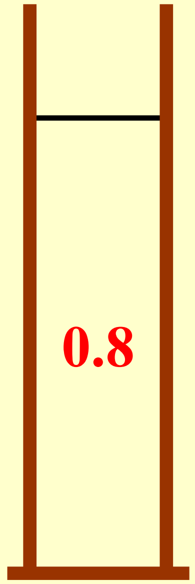
☞ **Off-line Algorithms**

View the entire item list before producing an answer.

Trouble-maker: The large items

Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.

〖Example〗 S_i = **0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1**



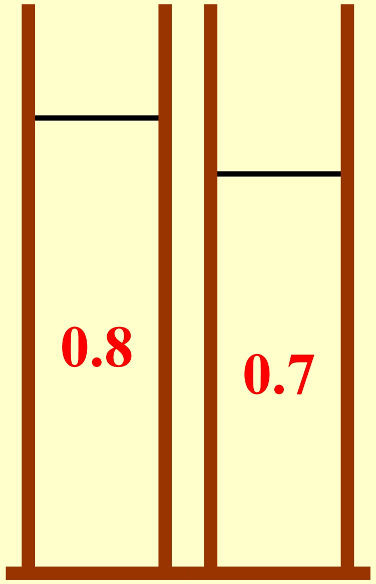
☞ **Off-line Algorithms**

View the entire item list before producing an answer.

Trouble-maker: The large items

Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.

〖Example〗 S_i = **0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1**



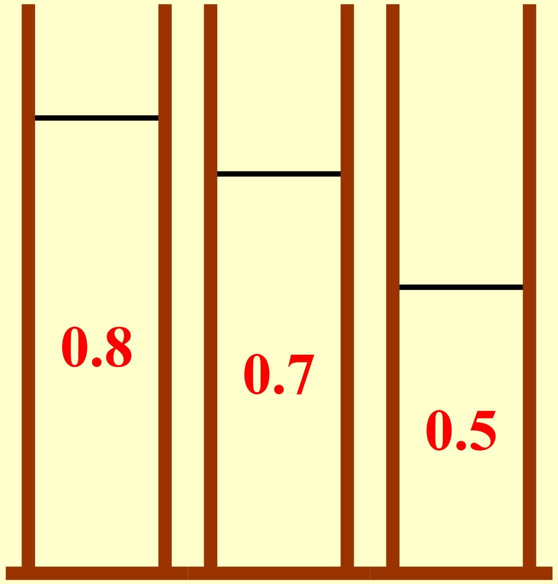
☞ **Off-line Algorithms**

View the entire item list before producing an answer.

Trouble-maker: The large items

Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.

〖Example〗 S_i = **0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1**



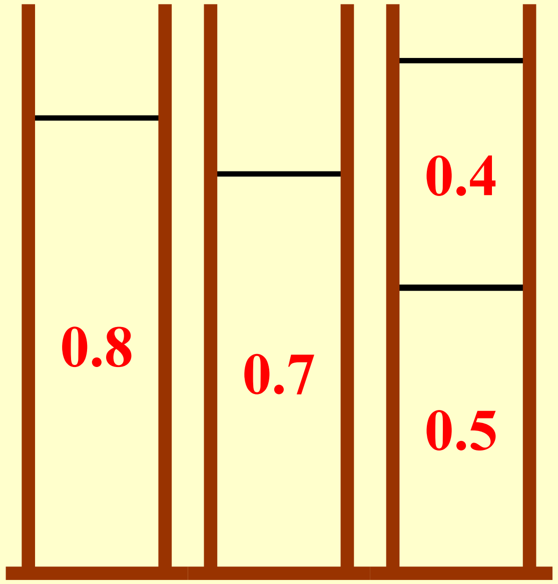
☞ Off-line Algorithms

View the entire item list before producing an answer.

Trouble-maker: The large items

Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – ***first*** (or ***best***) ***fit decreasing***.

〖Example〗 S_i = 0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1



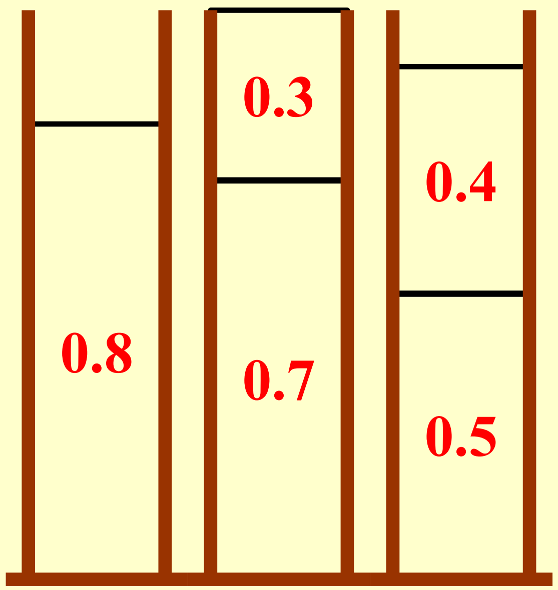
☞ Off-line Algorithms

View the entire item list before producing an answer.

Trouble-maker: The large items

Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.

〖Example〗 $S_i = 0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1$



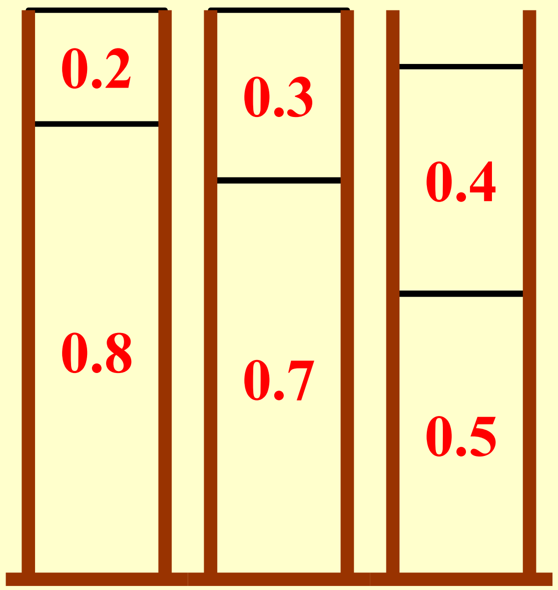
☞ Off-line Algorithms

View the entire item list before producing an answer.

Trouble-maker: The large items

Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.

〖Example〗 S_i **= 0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1**



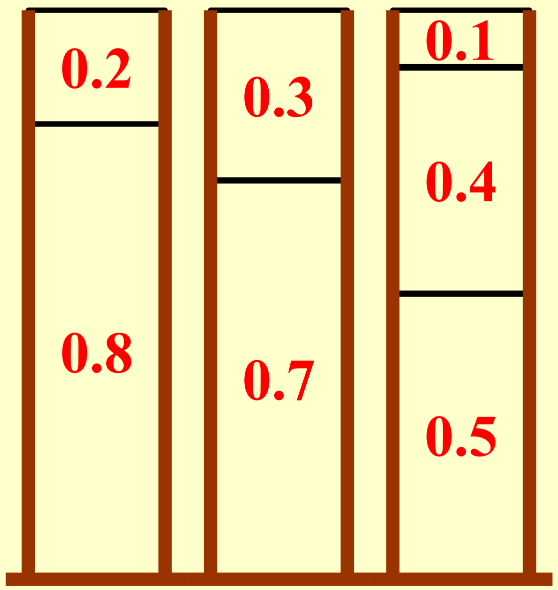
☞ Off-line Algorithms

View the entire item list before producing an answer.

Trouble-maker: The large items

Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.

〖Example〗 S_i = **0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1**



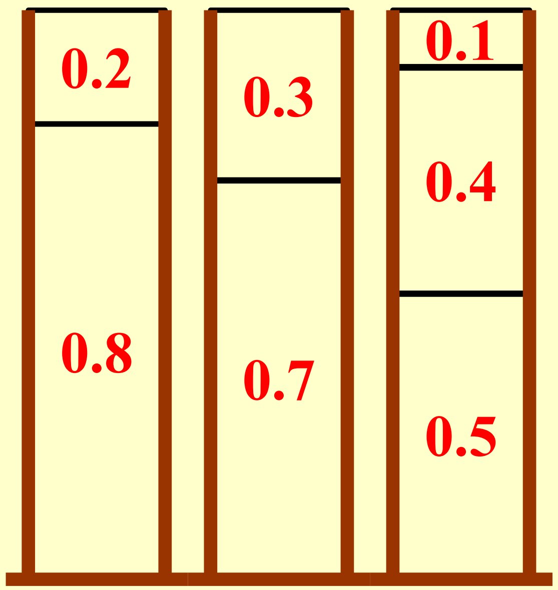
☞ **Off-line Algorithms**

View the entire item list before producing an answer.

Trouble-maker: The large items

Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.

〖Example〗 S_i = **0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1**



【Theorem】 Let M be the optimal number of bins required to pack a list I of items. Then *first fit decreasing* never uses more than $11M / 9 + 6/9$ bins. There exist sequences such that *first fit decreasing* uses $11M / 9 + 6/9$ bins.

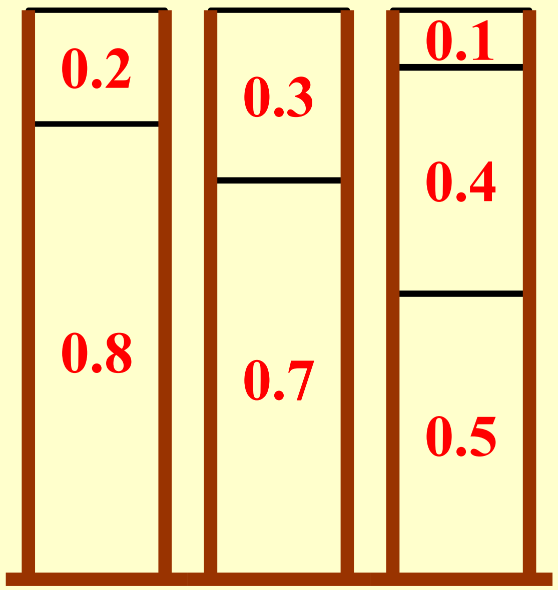
☞ Off-line Algorithms

View the entire item list before producing an answer.

Trouble-maker: The large items

Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.

〖Example〗 S_i = **0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1**



【Theorem】 Let M be the optimal number of bins required to pack a list I of items. Then *first fit decreasing* never uses more than $11M / 9 + 6/9$ bins. There exist sequences such that *first fit decreasing* uses $11M / 9 + 6/9$ bins.

Simple greedy heuristics can give good results.

Outline: Approximation Algorithms

- Bin packing
- 0-1 knapsack
- K-center selection
- Take-home messages

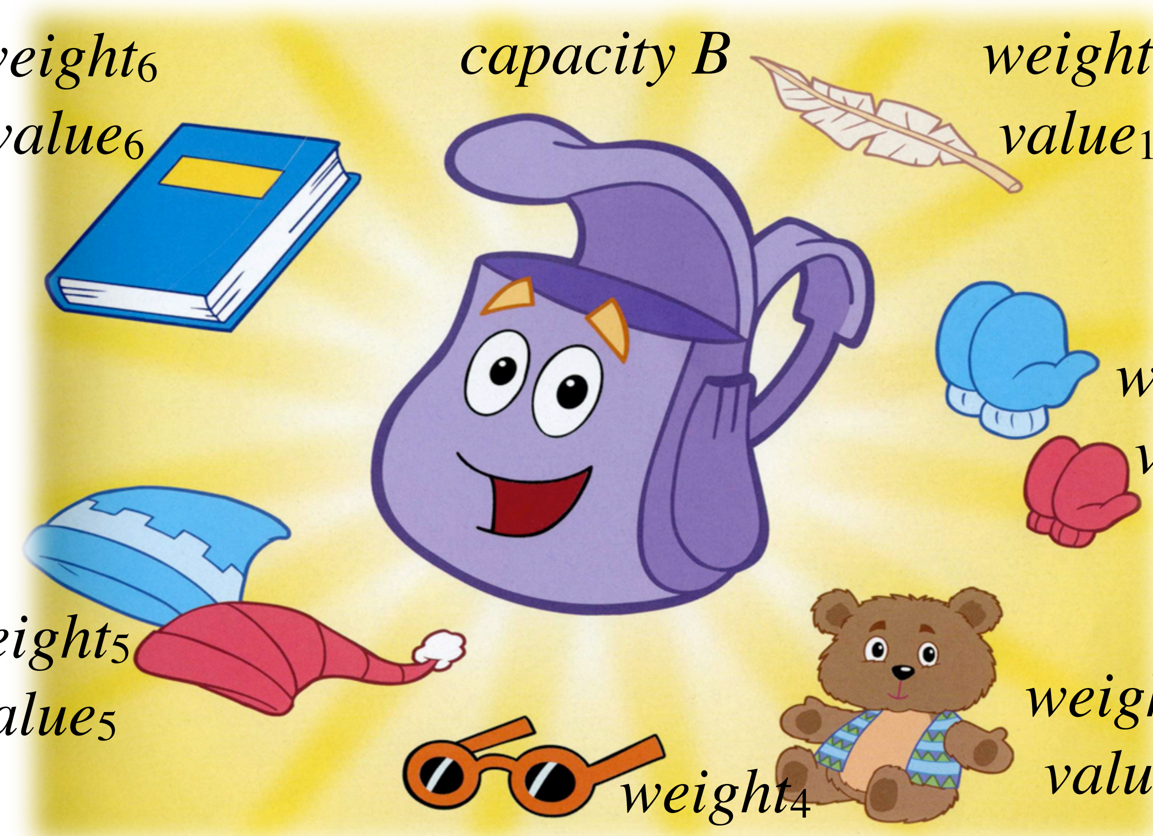
Knapsack Problem

Instance: n items i=1,2, …, n;

weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

Find a subset of items whose total weight is bounded by B and total value is maximized.



Knapsack Problem

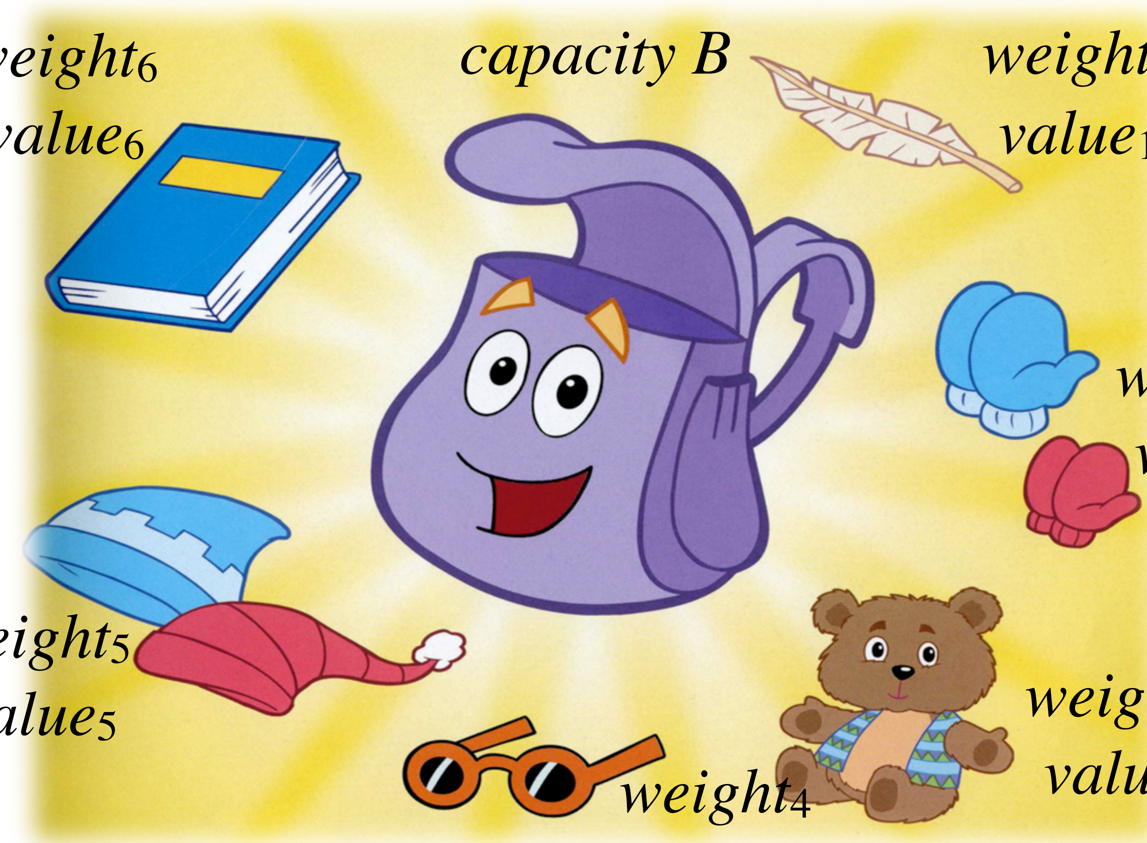
Instance: n items i=1,2, ..., n;

weights $w_1, w_2, ..., w_n \in \mathbb{Z}^+$; values $v_1, v_2, ..., v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

Find a subset of items whose total weight is bounded by B and total value is maximized.

- 0-1 Knapsack problem
- one of Karp's 21 **NP**-complete problems



Knapsack is NP-complete

KNAPSACK. Given a set X, weights $w_i \geq 0$, values $v_i \geq 0$, a weight limit W, and a target value V, is there a subset $S \subseteq X$ such that:

$$\sum_{i \in S} w_i \;\leq\; W$$
$$\sum_{i \in S} v_i \;\geq\; V$$

SUBSET-SUM. Given a set X, values $u_i \geq 0$, and an integer U, is there a subset $S \subseteq X$ whose elements sum to exactly U?

Theorem. SUBSET-SUM $\leq_P$ KNAPSACK.

Pf. Given instance $(u_1, \ldots, u_n, U)$ of SUBSET-SUM, create KNAPSACK instance:

$$v_i = w_i = u_i \qquad \sum_{i \in S} u_i \;\leq\; U$$
$$V = W = U \qquad \sum_{i \in S} u_i \;\geq\; U$$

Greedy *Heuristics*

Instance: n items i=1,2, ..., n;

weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

Find an $S \subseteq \{1,2, \ldots, n\}$ that maximizes $\sum_{i\in S} v_i$

subject to $\sum_{i\in S} w_i \leq B$.

Sort all items according to the ratio $r_i = v_i/w_i$

so that $r_1 \geq r_2 \geq \cdots \geq r_n$;

for i=1,2, ..., n

item i joins S if the resulting total weight $\leq B$;

approximation ratio: arbitrarily bad

Greedy *Heuristics*

Instance: n items i=1,2, ..., n;
weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;
knapsack capacity $B \in \mathbb{Z}^+$;

Find an $S \subseteq \{1,2, \ldots, n\}$ that maximizes $\sum_{i\in S} v_i$
subject to $\sum_{i\in S} w_i \le B$.

Sort all items according to the ratio $r_i = v_i/w_i$
so that $r_1 \ge r_2 \ge \cdots \ge r_n$;
for i=1,2, ..., n
item i joins S if the resulting total weight $\le B$;

approximation ratio: arbitrarily bad

Interestingly, we can have a 2-approximation algorithm with greedy technique.

The Knapsack Problem — fractional version

The Knapsack Problem — fractional version

A knapsack with a capacity M is to be packed. Given N items. Each item i has a weight w_i and a profit p_i. If x_i is the percentage of the item i being packed, then the packed profit will be $p_i x_i$.

The Knapsack Problem — fractional version

A knapsack with a capacity M is to be packed. Given N items. Each item i has a weight w_i and a profit p_i. If x_i is the percentage of the item i being packed, then the packed profit will be $p_i x_i$.

An optimal packing is a feasible one with maximum profit. That is, we are supposed to find the values of x_i such that $\sum_{i=1}^{n} p_i x_i$ obtains its maximum under the constraints

$$\sum_{i=1}^{n} w_i x_i \le M \quad \text{and} \quad x_i \in [0, 1] \text{ for } 1 \le i \le n$$

The Knapsack Problem — fractional version

A knapsack with a capacity M is to be packed. Given N items. Each item i has a weight w_i and a profit p_i. If x_i is the percentage of the item i being packed, then the packed profit will be $p_i x_i$.

An optimal packing is a feasible one with maximum profit. That is, we are supposed to find the values of x_i such that $\sum_{i=1}^{n} p_i x_i$ obtains its maximum under the constraints

$$\sum_{i=1}^{n} w_i x_i \leq M \quad \text{and} \quad x_i \in [0, 1] \ \text{for} \ 1 \leq i \leq n$$

Q: What must we do in each stage?

The Knapsack Problem — fractional version

A knapsack with a capacity M is to be packed. Given N items. Each item i has a weight w_i and a profit p_i. If x_i is the percentage of the item i being packed, then the packed profit will be $p_i x_i$.

An optimal packing is a feasible one with maximum profit. That is, we are supposed to find the values of x_i such that $\sum_{i=1}^{n} p_i x_i$ obtains its maximum under the constraints

$$\sum_{i=1}^{n} w_i x_i \le M \quad \text{and} \quad x_i \in [0,1] \text{ for } 1 \le i \le n$$

Q: What must we do in each stage?

A: Pack one item into the knapsack.

The Knapsack Problem — fractional version

A knapsack with a capacity M is to be packed. Given N items. Each item i has a weight w_i and a profit p_i. If x_i is the percentage of the item i being packed, then the packed profit will be $p_i x_i$.

An optimal packing is a feasible one with maximum profit. That is, we are supposed to find the values of x_i such that $\sum_{i=1}^{n} p_i x_i$ obtains its maximum under the constraints

$$\sum_{i=1}^{n} w_i x_i \le M \quad \text{and} \quad x_i \in [0, 1] \text{ for } 1 \le i \le n$$

Q: What must we do in each stage?

A: Pack one item into the knapsack.

Q: On which criterion shall we be greedy?

The Knapsack Problem — fractional version

A knapsack with a capacity M is to be packed. Given N items. Each item i has a weight w_i and a profit p_i. If x_i is the percentage of the item i being packed, then the packed profit will be $p_i x_i$.

An optimal packing is a feasible one with maximum profit. That is, we are supposed to find the values of x_i such that $\sum_{i=1}^{n} p_i x_i$ obtains its maximum under the constraints

$$\sum_{i=1}^{n} w_i x_i \leq M \quad \text{and} \quad x_i \in [0, 1] \text{ for } 1 \leq i \leq n$$

Q: What must we do in each stage?

A: Pack one item into the knapsack.

Q: On which criterion shall we be greedy?

① maximum profit

② minimum weight

③ maximum profit density p_i / w_i

The Knapsack Problem — fractional version

A knapsack with a capacity M is to be packed. Given N items. Each item i has a weight w_i and a profit p_i. If x_i is the percentage of the item i being packed, then the packed profit will be $p_i x_i$.

An optimal packing is a feasible one with maximum profit. That is, we are supposed to find the values of x_i such that $\sum_{i=1}^{n} p_i x_i$ obtains its maximum under the constraints

$$\sum_{i=1}^{n} w_i x_i \le M \quad \text{and} \quad x_i \in [0, 1] \text{ for } 1 \le i \le n$$

Q: What must we do in each stage?

A: Pack one item into the knapsack.

Q: On which criterion shall we be greedy?

① maximum profit

② minimum weight

③ maximum profit density p_i / w_i

The Knapsack Problem — fractional version

A knapsack with a capacity M is to be packed. Given N items. Each item i has a weight w_i and a profit p_i. If x_i is the percentage of the item i being packed, then the packed profit will be $p_i x_i$.

An optimal packing is a feasible one with maximum profit. That is, we are supposed to find the values of x_i such that $\sum_{i=1}^{n} p_i x_i$ obtains its maximum under the constraints

$$\sum_{i=1}^{n} w_i x_i \le M \quad \text{and} \quad x_i \in [0, 1] \text{ for } 1 \le i \le n$$

Q: What must we do in each stage?

A: Pack one item into the knapsack.

Q: On which criterion shall we be greedy?

① maximum profit

② minimum weight

③ maximum profit density p_i / w_i

Example :
$n = 3$, $M = 20$,
$(p1, p2, p3) = (25, 24, 15)$
$(w1, w2, w3) = (18, 15, 10)$
Solution is...?

The Knapsack Problem — fractional version

A knapsack with a capacity M is to be packed. Given N items. Each item i has a weight w_i and a profit p_i. If x_i is the percentage of the item i being packed, then the packed profit will be $p_i x_i$.

An optimal packing is a feasible one with maximum profit. That is, we are supposed to find the values of x_i such that $\sum_{i=1}^{n} p_i x_i$ obtains its maximum under the constraints

$$\sum_{i=1}^{n} w_i x_i \le M \quad \text{and} \quad x_i \in [0,1] \ \text{for} \ 1 \le i \le n$$

Q: What must we do in each stage?

A: Pack one item into the knapsack.

Q: On which criterion shall we be greedy?

① maximum profit

② minimum weight

③ maximum profit density p_i / w_i

Example :
$n = 3$, $M = 20$,
$(p1, p2, p3) = (25, 24, 15)$
$(w1, w2, w3) = (18, 15, 10)$
Solution is...?
(0, 1, 1/2)
P = 31.5

The Knapsack Problem — 0-1 version

The Knapsack Problem — 0-1 version

x_i is either 1 or 0

The Knapsack Problem — 0-1 version

NP-hard

The Knapsack Problem — 0-1 version

NP-hard

Example :

$n = 5$, $M = 11$,

$(p_1, p_2, p_3, p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2, 5, 6, 7)$

Solution is...?

The Knapsack Problem — 0-1 version

NP-hard

Example :

$n = 5$, $M = 11$,

$(p_1, p_2, p_3, p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2, 5, 6, 7)$

Solution is...? (0, 0, 1, 1, 0)

P = 40

The Knapsack Problem — 0-1 version

NP-hard

Example :

$n = 5$, $M = 11$,

$(p_1, p_2, p_3, p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2, 5, 6, 7)$

The greedy solution is

Solution is...? (0, 0, 1, 1, 0)

P = 40

The Knapsack Problem — 0-1 version

NP-hard

Example :

$n = 5$, $M = 11$,

$(p_1, p_2, p_3, p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2, 5, 6, 7)$

Solution is...? (0, 0, 1, 1, 0)

P = 40

The greedy solution is

(1, 1, 0, 0, 1)

P = 35

The Knapsack Problem — 0-1 version

NP-hard

Example :

$n = 5$, $M = 11$,

$(p_1, p_2, p_3, p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2, 5, 6, 7)$

Solution is...? (0, 0, 1, 1, 0)
P = 40

The greedy solution is

(1, 1, 0, 0, 1)
P = 35

What if we run both maximum profit greedy and profit density greedy, then output the better result?

The Knapsack Problem — 0-1 version

NP-hard

Example :

$n = 5$, $M = 11$,

$(p_1, p_2, p_3, p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2, 5, 6, 7)$

Solution is...? (0, 0, 1, 1, 0)
P = 40

The greedy solution is
(1, 1, 0, 0, 1)
P = 35

What if we run both maximum profit greedy and profit density greedy, then output the better result?

The approximation ratio is 2.

The Knapsack Problem — 0-1 version

NP-hard

Example :

$n = 5$, $M = 11$,

$(p_1, p_2, p_3, p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2, 5, 6, 7)$

Solution is...? (0, 0, 1, 1, 0) P = 40

The greedy solution is (1, 1, 0, 0, 1) P = 35

What if we run both maximum profit greedy and profit density greedy, then output the better result?

The approximation ratio is 2.

Proof:

The Knapsack Problem — 0-1 version

NP-hard

Example :

$n = 5$, $M = 11$,

$(p_1, p_2, p_3, p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2, 5, 6, 7)$

Solution is...? (0, 0, 1, 1, 0) P = 40

The greedy solution is (1, 1, 0, 0, 1) P = 35

What if we run both maximum profit greedy and profit density greedy, then output the better result?

The approximation ratio is 2.

Proof: $p_{max} \le P_{opt} \le P_{pd,\ frac}$

The Knapsack Problem — 0-1 version

NP-hard

Example :

$n = 5$, $M = 11$,

$(p_1, p_2, p_3, p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2, 5, 6, 7)$

Solution is...? (0, 0, 1, 1, 0) P = 40

The greedy solution is (1, 1, 0, 0, 1) P = 35

What if we run both maximum profit greedy and profit density greedy, then output the better result?

The approximation ratio is 2.

Proof: $p_{max} \le P_{opt} \le P_{pd,\ frac}$

$p_{max} \le P_{mp}$

The Knapsack Problem — 0-1 version

NP-hard

Example :

$n = 5,\ M = 11,$

$(p_1,\ p_2,\ p_3,\ p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2,\ 5,\ 6,\ 7)$

Solution is...? (0, 0, 1, 1, 0)
P = 40

The greedy solution is (1, 1, 0, 0, 1)
P = 35

What if we run both maximum profit greedy and profit density greedy, then output the better result?

The approximation ratio is 2.

Proof: $p_{max} \le P_{opt} \le P_{pd,\ frac}$

$p_{max} \le P_{mp}$

$P_{opt} \le P_{pd,\ frac} \le P_{pd,\ 0\text{-}1} + p_{max} \le P_{pd,\ 0\text{-}1} + P_{mp}$

The Knapsack Problem — 0-1 version

NP-hard

Example :

$n = 5$, $M = 11$,

$(p_1, p_2, p_3, p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2, 5, 6, 7)$

Solution is...? (0, 0, 1, 1, 0)
P = 40

The greedy solution is (1, 1, 0, 0, 1)
P = 35

What if we run both maximum profit greedy and profit density greedy, then output the better result?

The approximation ratio is 2.

Proof: $p_{max} \le P_{opt} \le P_{pd,\,frac}$

$p_{max} \le P_{mp}$

$P_{opt} \le P_{pd,\,frac} \le P_{pd,\,0\text{-}1} + p_{max} \le P_{pd,\,0\text{-}1} + P_{mp}$

Check https://www.cs.cornell.edu/courses/cs4820/2018fa/lectures/approx-onlinet.pdf

☞ **A Dynamic Programming Solution**

☞ A Dynamic Programming Solution

$W_{i,\,p}$ = **the minimum weight of a collection from {1, …,** i **} with total profit being exactly** p

☞ A Dynamic Programming Solution

$W_{i,p}$ = the minimum weight of a collection from {1, …, i } with total profit being exactly p

① take i :

☞ A Dynamic Programming Solution

$W_{i,p}$ = the minimum weight of a collection from {1, …, i } with total profit being exactly p

① take i : $W_{i,p} = w_i + W_{i-1,p-p_i}$

☞ A Dynamic Programming Solution

$W_{i,p}$ = the minimum weight of a collection from {1, …, i } with total profit being exactly p

① take i : $W_{i,p} = w_i + W_{i-1,p-p_i}$

② skip i :

☞ A Dynamic Programming Solution

$W_{i,p}$ = the minimum weight of a collection from {1, …, i } with total profit being exactly p

① take i : $W_{i,p} = w_i + W_{i-1,p-p_i}$

② skip i : $W_{i,p} = W_{i-1,p}$

☞ A Dynamic Programming Solution

$W_{i,p}$ = the minimum weight of a collection from {1, …, i } with total profit being exactly p

① take i : $W_{i,p} = w_i + W_{i-1,p-p_i}$

② skip i : $W_{i,p} = W_{i-1,p}$

③ impossible to get p :

☞ A Dynamic Programming Solution

$W_{i,p}$ = the minimum weight of a collection from {1, …, i } with total profit being exactly p

① take i : $W_{i,p} = w_i + W_{i-1,\,p-p_i}$

② skip i : $W_{i,p} = W_{i-1,\,p}$

③ impossible to get p : $W_{i,p} = \infty$

☞ A Dynamic Programming Solution

$W_{i,p}$ = the minimum weight of a collection from {1, …, i } with total profit being exactly p

① take i : $W_{i,p} = w_i + W_{i-1,p-p_i}$

② skip i : $W_{i,p} = W_{i-1,p}$

③ impossible to get p : $W_{i,p} = \infty$

$$W_{i,p} = \begin{cases} \infty & i = 0 \\ W_{i-1,p} & p_i > p \\ \min\{W_{i-1,p},\, w_i + W_{i-1,p-p_i}\} & \textit{otherwise} \end{cases}$$

☞ A Dynamic Programming Solution

$W_{i,p}$ = the minimum weight of a collection from {1, …, i } with total profit being exactly p

① take i : $W_{i,p} = w_i + W_{i-1,p-p_i}$

② skip i : $W_{i,p} = W_{i-1,p}$

③ impossible to get p : $W_{i,p} = \infty$

$$W_{i,p} = \begin{cases} \infty & i = 0 \\ W_{i-1,p} & p_i > p \\ \min\{W_{i-1,p}, w_i + W_{i-1,p-p_i}\} & otherwise \end{cases}$$

i = 1, …, n; p = 1, …, ?

☞ A Dynamic Programming Solution

$W_{i,p}$ = the minimum weight of a collection from {1, …, *i* } with total profit being exactly p

① take i : $W_{i,p} = w_i + W_{i-1,p-p_i}$

② skip i : $W_{i,p} = W_{i-1,p}$

③ impossible to get p : $W_{i,p} = \infty$

$$W_{i,p} = \begin{cases} \infty & i = 0 \\ W_{i-1,p} & p_i > p \\ \min\{W_{i-1,p}, w_i + W_{i-1,p-p_i}\} & otherwise \end{cases}$$

$i = 1, \dots, n;\ p = 1, \dots,\ n\, p_{max}$

☞ A Dynamic Programming Solution

$W_{i,p}$ = the minimum weight of a collection from {1, …, i } with total profit being exactly p

① take i : $W_{i,p} = w_i + W_{i-1,p-p_i}$

② skip i : $W_{i,p} = W_{i-1,p}$

③ impossible to get p : $W_{i,p} = \infty$

$$W_{i,p} = \begin{cases} \infty & i = 0 \\ W_{i-1,p} & p_i > p \\ \min\{ W_{i-1,p}, w_i + W_{i-1,p-p_i} \} & otherwise \end{cases}$$

$i = 1, \ldots, n;\ p = 1, \ldots, n\, p_{max}$ ➡ $O(n^2 p_{max})$

☞ **What if p_{max} is LARGE?**

☞ What if p_{max} is LARGE?

Item	Profit	Weight
1	134,221	1
2	656,342	2
3	1,810,013	5
4	22,217,800	6
5	28,343,199	7

M = 11

☞ **What if p_{max} is LARGE?**

Item	Profit	Weight
1	134,221	1
2	656,342	2
3	1,810,013	5
4	22,217,800	6
5	28,343,199	7

M = 11

☝ **Round all profit values up to lie in smaller range!**

☞ What if p_{max} is LARGE?

Item	Profit	Weight
1	134,221	1
2	656,342	2
3	1,810,013	5
4	22,217,800	6
5	28,343,199	7

M = 11

Item	Profit	Weight
1	2	1
2	7	2
3	19	5
4	223	6
5	284	7

M = 11

☝ **Round all profit values up to lie in smaller range!**

☞ What if p_{max} is LARGE?

Item	Profit	Weight
1	134,221	1
2	656,342	2
3	1,810,013	5
4	22,217,800	6
5	28,343,199	7

M = 11

Item	Profit	Weight
1	2	1
2	7	2
3	19	5
4	223	6
5	284	7

M = 11

☝ **Round all profit values up to lie in smaller range!**

$(1+\varepsilon)\, P_{alg} \le P$ **for any feasible solution P**

☞ **What if p_{max} is LARGE?**

Item	Profit	Weight
1	134,221	1
2	656,342	2
3	1,810,013	5
4	22,217,800	6
5	28,343,199	7

M = 11

Item	Profit	Weight
1	2	1
2	7	2
3	19	5
4	223	6
5	284	7

M = 11

☝ **Round all profit values up to lie in smaller range!**

$(1+\varepsilon)\, P_{alg} \leq P$ **for any feasible solution P**

precision parameter

Dynamic Programming

Instance: n items i=1,2, ..., n;
weights $w_1, w_2, ..., w_n \in \mathbb{Z}^+$; values $v_1, v_2, ..., v_n \in \mathbb{Z}^+$;
knapsack capacity $B \in \mathbb{Z}^+$;

Find a subset of items whose total weight is bounded by B and total value is maximized.

$A(i, v)$ = minimum total weight of $S \subseteq \{1,2, ..., i\}$ with total value *exactly* v

$$A(i, v) = \min\{ A(i-1, v), \; A(i-1, v-v_i) + w_i \}$$

$$A(1, v) = \begin{cases} w_1 & \text{if } v = v_1 \\ \infty & \text{otherwise} \end{cases}$$

knapsack: $\max\{v: A(n,v) \le B\}$

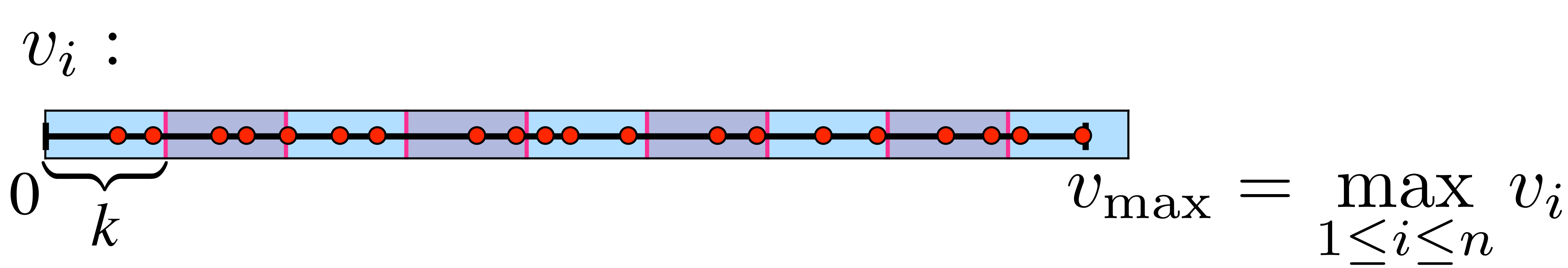
Dynamic programming:
time complexity $O(nV)$
where $V = \sum_i v_i$

Pseudo-Polynomial Time!

Scaling & Rounding

Instance: n items i=1,2, ..., n;
weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;
knapsack capacity $B \in \mathbb{Z}^+$;

Set k = (to be fixed) ;
for i=1,2, ..., n, let $v'_i = \lfloor v_i / k \rfloor$;
return the knapsack solution found by dynamic programming with new values v'_i ;



Scaling & Rounding

Instance: n items i=1,2, ..., n;
weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;
knapsack capacity $B \in \mathbb{Z}^+$;

Set k = (to be fixed) ;
for i=1,2, ..., n, let $v'_i = \lfloor v_i / k \rfloor$;
return the knapsack solution found by dynamic programming with new values v'_i ;

time complexity: $O(n V') = O(nV/k)$

where $V' = \sum_i v'_i = \sum_i \lfloor v_i / k \rfloor = O(V/k)$

and $V = \sum_i v_i$

Instance: n items i=1,2, ..., n;
weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;
knapsack capacity $B \in \mathbb{Z}^+$;

Set k = (to be fixed);
for i=1,2, ..., n, let $v'_i = \lfloor v_i / k \rfloor$;
return the knapsack solution found by dynamic programming with new values v'_i;

time complexity: $O(nV/k)$ where $V = \sum_i v_i$

S^*: optimal knapsack solution of the original instance

$$OPT = \sum_{i\in S^*} v_i = k\sum_{i\in S^*}\frac{v_i}{k} \le k\sum_{i\in S^*}\left(\left\lfloor\frac{v_i}{k}\right\rfloor + 1\right) \le k\sum_{i\in S^*}\left\lfloor\frac{v_i}{k}\right\rfloor + nk$$

S: the solution returned by the algorithm
(optimal solution of the scaled instance) ⇒ $\sum_{i\in S}\left\lfloor\frac{v_i}{k}\right\rfloor \ge \sum_{i\in S^*}\left\lfloor\frac{v_i}{k}\right\rfloor$

$$SOL = \sum_{i\in S} v_i \ge k\sum_{i\in S}\left\lfloor\frac{v_i}{k}\right\rfloor \ge k\sum_{i\in S^*}\left\lfloor\frac{v_i}{k}\right\rfloor \ge OPT - nk$$

Instance: n items $i=1,2, \ldots, n$;
weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;
knapsack capacity $B \in \mathbb{Z}^+$;

Set k = (to be fixed) ;
for $i=1,2, \ldots, n$, let $v'_i = \lfloor v_i / k \rfloor$;
return the knapsack solution found by
dynamic programming with new values v'_i ;

time complexity: $O(nV/k)$ where $V = \sum_i v_i \leq n v_{\max}$

OPT: optimal value of the original instance

SOL: value of the solution returned by the algorithm

$SOL \geq OPT - nk$ $\Rightarrow$ $$\frac{SOL}{OPT} \geq 1 - \frac{nk}{OPT} \geq 1 - \frac{nk}{v_{\max}}$$

WLOG: $$OPT \geq v_{\max} = \max_{1 \leq i \leq n} v_i$$

Instance: n items i=1,2, ..., n;

weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

for any $0 \le \varepsilon \le 1$:

Set $k = \left\lfloor \frac{\epsilon v_{\max}}{n} \right\rfloor$; where $v_{\max} = \max_{1 \le i \le n} v_i$

for i=1,2, ..., n, let $v'_i = \lfloor v_i / k \rfloor$;

return the knapsack solution found by dynamic programming with new values v'_i;

time complexity: $O\left(\frac{n^2 v_{\max}}{k}\right) = O\left(\frac{n^3}{\epsilon}\right)$

OPT: optimal value of the original instance

SOL: value of the solution returned by the algorithm

$$\frac{SOL}{OPT} \ge 1 - \frac{nk}{v_{\max}} \ge 1 - \epsilon$$

Approximation Ratio

Optimization problem:

- instance I:

  $\mathrm{OPT}(I)$ = optimum of instance I

- algorithm A: returns a solution s for every instance I

  $\mathrm{SOL}_A(I)$ = value returned by A on instance I

minimization: approximation ratio of algorithm A is α

if $\forall$ instance I : $\dfrac{\mathrm{SOL}_A(I)}{\mathrm{OPT}(I)} \leq \alpha$

maximization: approximation ratio of algorithm A is α

if $\forall$ instance I : $\dfrac{OPT(I)}{SOL_A(I)} \leq \alpha$

ε-approximation: $(1-\varepsilon)\,\mathrm{OPT}(I) \leq \mathrm{SOL}_A(I) \leq (1+\varepsilon)\,\mathrm{OPT}(I)$

(maximization) (minimization)

Approximation Ratio

Optimization problem:

- instance I:

  $\mathrm{OPT}(I)$ = optimum of instance I

- algorithm A: returns a solution s for every instance I and $0 \leq \varepsilon \leq 1$

$\mathrm{SOL}_A(\varepsilon, I)$ = value returned by A on instance I and ε

- A is a Polynomial-Time Approximation Scheme (PTAS) if:

  $\forall 0 \leq \varepsilon \leq 1$, A returns in polynomial time and

$$(1\text{-}\varepsilon)\ \mathrm{OPT}(I) \leq \mathrm{SOL}_A(\varepsilon, I) \leq (1+\varepsilon)\ \mathrm{OPT}(I)$$

(maximization) (minimization)

- A is a *Fully* Polynomial-Time Approximation Scheme (FPTAS) if:

  furthermore, A returns in time $\mathrm{Poly}(1/\varepsilon, n)$ where $n = |I|$

  (in *binary* code)

Approximation Ratio

Optimization problem:

- instance I:

  $\mathrm{OPT}(I)$ = optimum of instance I

- algorithm A: returns a solution s for every instance I and $0 \leq \varepsilon \leq 1$

$\mathrm{SOL}_A(\varepsilon, I)$ = value returned by A on instance I and ε

- A is a Polynomial-Time Approximation Scheme (PTAS) if:

  $\forall 0 \leq \varepsilon \leq 1$, A returns in polynomial time and

$$(1-\varepsilon)\,\mathrm{OPT}(I) \leq \mathrm{SOL}_A(\varepsilon, I) \leq (1+\varepsilon)\,\mathrm{OPT}(I)$$

(maximization) (minimization)

- A is a *Fully* Polynomial-Time Approximation Scheme (FPTAS) if:

  furthermore, A returns in time $\mathrm{Poly}(1/\varepsilon, n)$ where $n = |I|$

(in *binary* code)

$O(n^{\frac{1}{\epsilon}})$ PTAS but not FPTAS $O(\frac{n}{\epsilon^2})$ PTAS and FPTAS

Instance: n items i=1,2, ..., n;

weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

for any $0 \le \varepsilon \le 1$:

Set $k = \left\lfloor \frac{\epsilon v_{\max}}{n} \right\rfloor$; where $v_{\max} = \max_{1 \le i \le n} v_i$

for i=1,2, ..., n, let $v'_i = \lfloor v_i / k \rfloor$;

return the knapsack solution found by dynamic programming with new values v'_i;

time complexity: $O\left(\frac{n^3}{\epsilon}\right)$

approximation ratio: $\frac{SOL}{OPT} \ge 1 - \epsilon$

FPTAS

Are FPTASs the "best" approximation algorithms?

A Nearly Quadratic-Time FPTAS for Knapsack

Lin Chen*
Zhejiang University
Hangzhou, China
chenlin198662@zju.edu.cn

Jiayi Lian†
Zhejiang University
Hangzhou, China
jiayilian@zju.edu.cn

Yuchen Mao‡
Zhejiang University
Hangzhou, China
maoyc@zju.edu.cn

Guochuan Zhang§
Zhejiang University
Hangzhou, China
zgc@zju.edu.cn

ABSTRACT

We investigate the classic Knapsack problem and propose a fully polynomial-time approximation scheme (FPTAS) that runs in $\widetilde{O}(n+(1/\varepsilon)^2)$ time. Prior to our work, the best running time is $\widetilde{O}(n+(1/\varepsilon)^{11/5})$ [Deng, Jin, and Mao'23]. Our algorithm is the best possible (up to a polylogarithmic factor), as Knapsack has no $O((n+1/\varepsilon)^{2-\delta})$-time FPTAS for any constant $\delta > 0$, conditioned on the conjecture that $(\min,+)$-convolution has no truly subquadratic-time algorithm.

Knapsack is a fundamental problem in combinatorial optimization and belongs to Karp's 21 NP-complete problems [21]. Consequently, extensive effort has been devoted to developing approximation algorithms for Knapsack. Knapsack admits a fully polynomial-time approximation scheme (FPTAS), which is an algorithm that takes a precision parameter ε and produces a solution that is within a factor $1+\varepsilon$ from the optimum in time poly$(n, 1/\varepsilon)$. Since the first FPTAS for Knapsack, there has been a long line of research on improving its running time, as summarized in Table 1.

STOC 2024: Proceedings of the 56th Annual ACM Symposium on Theory of Computing • Pages 283 - 294
https://doi.org/10.1145/3618260.3649730

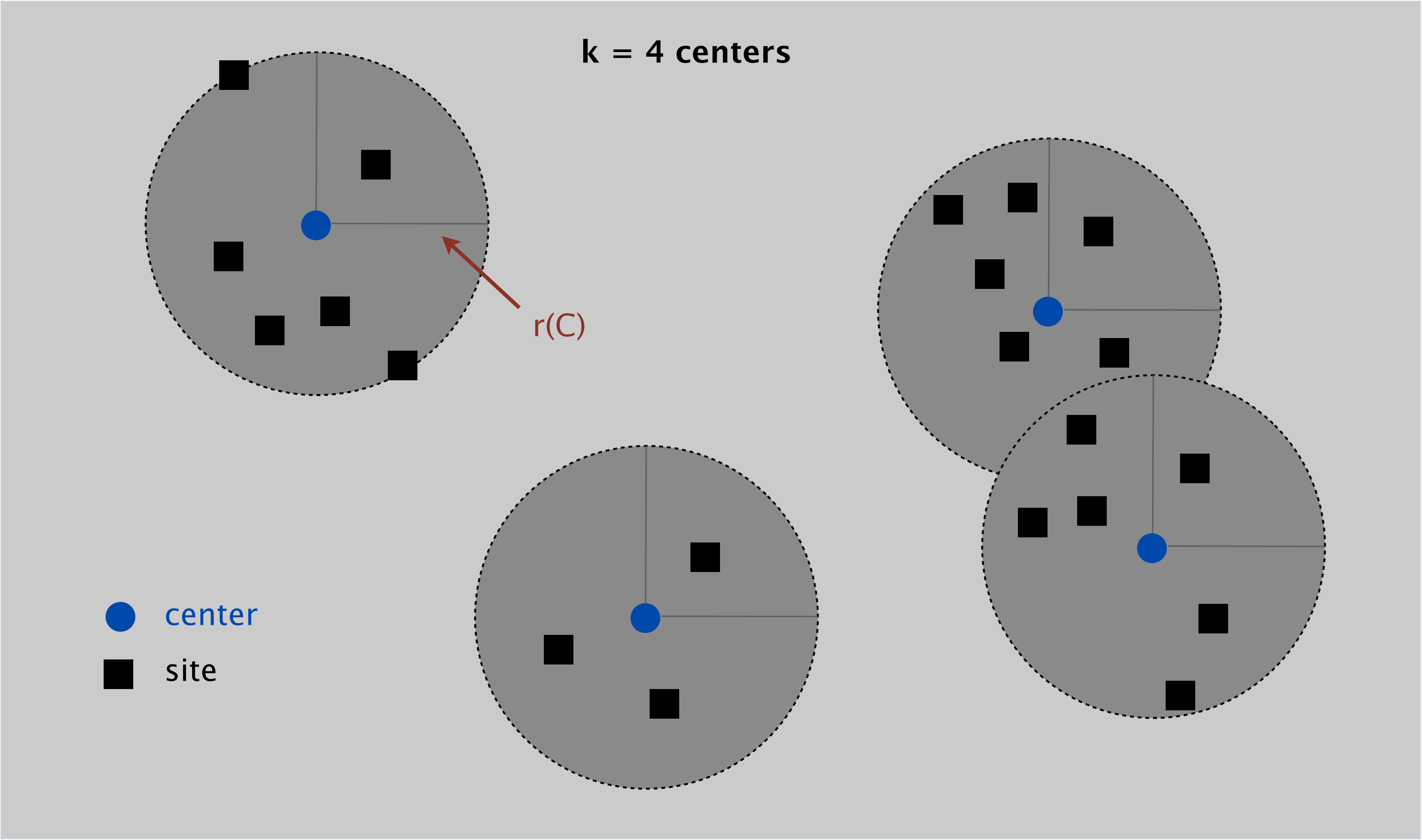
Outline: Approximation Algorithms

- Bin packing
- 0-1 knapsack
- K-center selection
- Take-home messages

Center selection problem

Input. Set of n sites $s_1, \ldots, s_n$ and an integer $k > 0$.

Center selection problem. Select set of k centers C so that maximum distance $r(C)$ from a site to nearest center is minimized.



Center selection problem

Input. Set of n sites $s_1, \ldots, s_n$ and an integer $k > 0$.

Center selection problem. Select set of k centers C so that maximum distance $r(C)$ from a site to nearest center is minimized.

Center selection problem

Input. Set of n sites $s_1, \ldots, s_n$ and an integer $k > 0$.

Center selection problem. Select set of k centers C so that maximum distance $r(C)$ from a site to nearest center is minimized.

Notation.

- $dist(x, y) =$ distance between sites x and y.
- $dist(s_i, C) = \min_{c \in C} dist(s_i, c)$ = distance from s_i to closest center.
- $r(C) = \max_i dist(s_i, C) =$ smallest covering radius.

Goal. Find set of centers C that minimizes $r(C)$, subject to $|C| = k$.

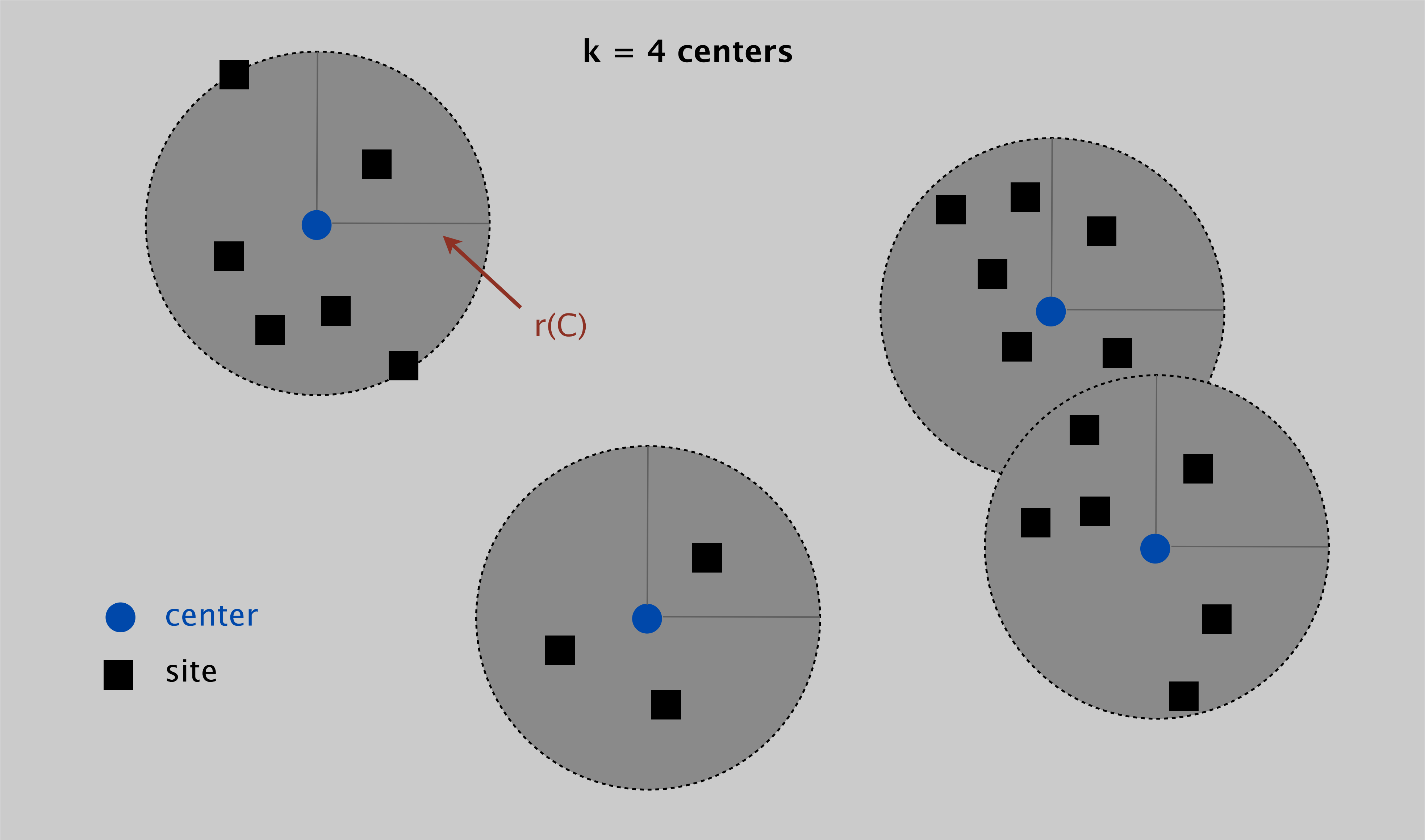
Distance function properties.

- $dist(x, x) = 0$ [identity]
- $dist(x, y) = dist(y, x)$ [symmetry]
- $dist(x, y) \leq dist(x, z) + dist(z, y)$ [triangle inequality]

Center selection example

Ex: each site is a point in the plane, a center can be any point in the plane, $dist(x, y)$ = Euclidean distance.

Remark: search can be infinite!



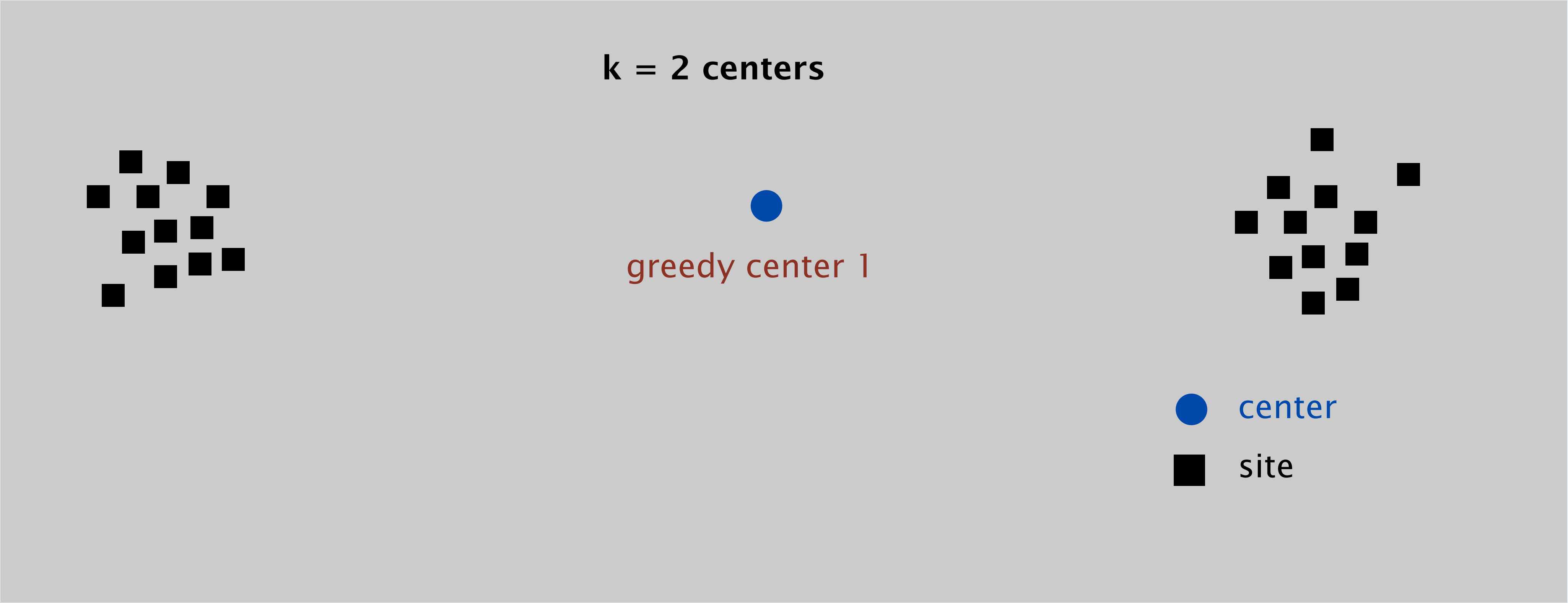
Greedy algorithm: a false start

Greedy algorithm. Put the first center at the best possible location for a single center, and then keep adding centers so as to reduce the covering radius each time by as much as possible.

Greedy algorithm: a false start

Greedy algorithm. Put the first center at the best possible location for a single center, and then keep adding centers so as to reduce the covering radius each time by as much as possible.

Remark: arbitrarily bad!



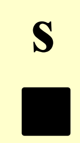
☞ **A Greedy Solution — try again …**

☞ **A Greedy Solution — try again …**

What if we know that $r(C^*) \le r$ where C^* is the optimal solution set?

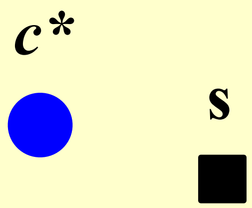
☞ A Greedy Solution — try again …

What if we know that $r(C^*) \le r$ where C^* is the optimal solution set?



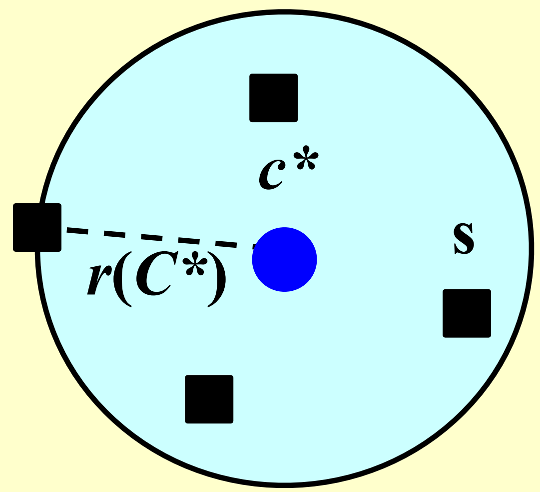
☞ A Greedy Solution — try again …

What if we know that $r(C^*) \le r$ where C^* is the optimal solution set?



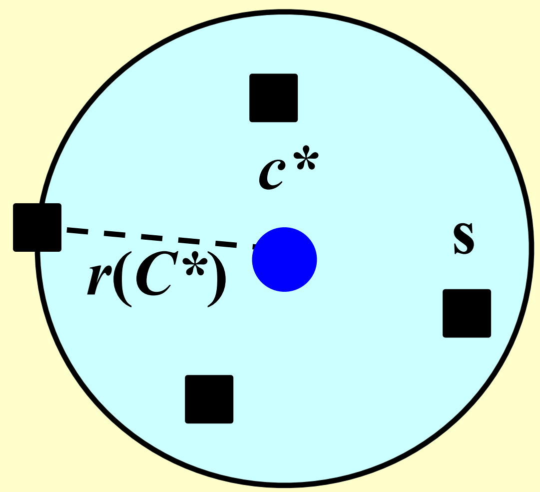
☞ **A Greedy Solution — try again …**

What if we know that $r(C^*) \le r$ where C^* is the optimal solution set?



☞ A Greedy Solution — try again …

What if we know that $r(C^*) \le r$ where C^* is the optimal solution set?

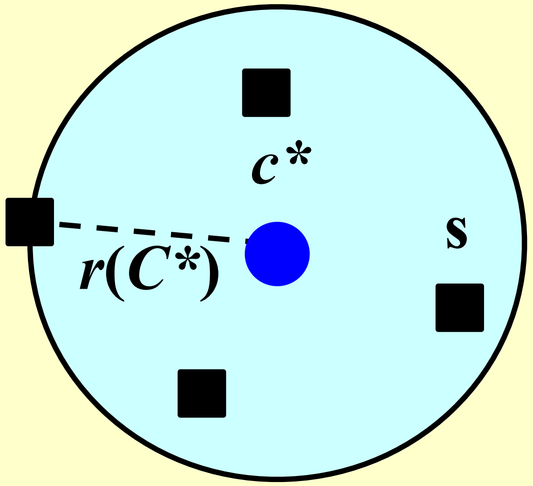


Discussion 15:

Take s to be the center, how can we select r so that s can cover all the sites that are covered by c^*?

☞ A Greedy Solution — try again …

What if we know that $r(C^*) \le r$ where C^* is the optimal solution set?



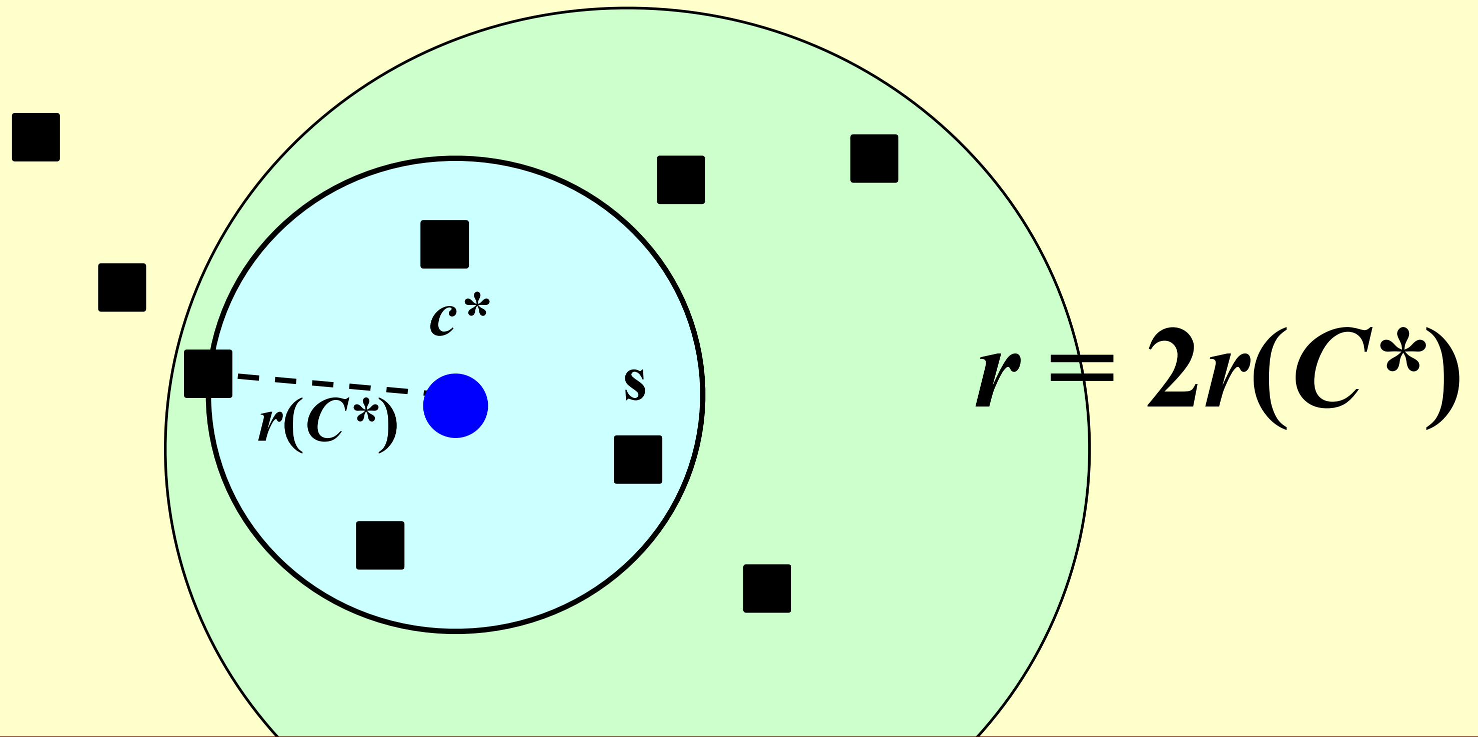
$$r = 2r(C^*)$$

Discussion 15:

Take s to be the center, how can we select r so that s can cover all the sites that are covered by c^*?

☞ A Greedy Solution — try again …

What if we know that $r(C^*) \le r$ where C^* is the optimal solution set?



Discussion 15:

Take s to be the center, how can we select r so that s can cover all the sites that are covered by c^*?

```
Centers  Greedy-2r ( Sites S[ ], int n, int K, double r*)
{   Sites  S'[ ] = S[ ]; /* S' is the set of the remaining sites */
    Centers  C[ ] = ∅;
    while ( S'[ ] != ∅ ) {
        Select any s from S' and add it to C;
        Delete all s' from S' that are at dist(s', s) ≤ 2r*;
    } /* end-while */
    if ( |C| ≤ K ) return C;
    else ERROR(No set of K centers with covering radius at most r*);
}
```

```
Centers  Greedy-2r ( Sites S[ ], int n, int K, double r*)
{   Sites  S'[ ] = S[ ]; /* S' is the set of the remaining sites */
    Centers  C[ ] = ∅;
    while ( S'[ ] != ∅ ) {
        Select any s from S' and add it to C;
        Delete all s' from S' that are at dist(s', s) ≤ 2r*;
    } /* end-while */
    if ( |C| ≤ K ) return C;
    else ERROR(No set of K centers with covering radius at most r*);
}
```

【Theorem】 Suppose the algorithm selects more than K centers. Then for any set C of size at most K, the covering radius is $r(C) > r^*$.

```
Centers  Greedy-2r ( Sites S[ ], int n, int K, double r*)
{   Sites  S'[ ] = S[ ]; /* S' is the set of the remaining sites */
    Centers  C[ ] = ∅;
    while ( S'[ ] != ∅ ) {
        Select any s from S' and add it to C;
        Delete all s' from S' that are at dist(s', s) ≤ 2r*;
    } /* end-while */
    if ( |C| ≤ K ) return C;
    else ERROR(No set of K centers with covering radius at most r*);
}
```

【Theorem】 Suppose the algorithm selects more than K centers. Then for any set C of size at most K, the covering radius is $r(C) > r^*$.

Proof see Algorithm Design Sec 11.2

Do we really know $r(C^*)$?

Do we really know $r(C^*)$?

- **Binary search for *r***

Do we really know $r(C^*)$?

Binary search for r

$0 < r \leq r_{max}$

Do we really know $r(C^*)$?

Binary search for r

$0 < r \le r_{max}$ **Guess:** $r = (0 + r_{max}) / 2$

Do we really know $r(C^*)$?

Binary search for r

$0 < r \le r_{max}$ **Guess:** $r = (0 + r_{max}) / 2$

Yes: K centers found with $2r$

or

No: r is too small

Do we really know $r(C^*)$?

Binary search for r

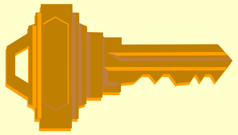
$0 < r \le r_{max}$ **Guess:** $r = (0 + r_{max}) / 2$

Yes: K centers found with $2r$

or

No: r is too small

Do we really know $r(C^*)$?



Binary search for r

$0 < r \le r_{max}$ **Guess:** $r = (\, 0 + r_{max} \,) / 2$

$$\left\{ \begin{array}{l} \textbf{Yes: } K \textbf{ centers found with } 2r \\ \textbf{or} \\ \textbf{No: } r \textbf{ is too small} \end{array} \right.$$

Do we really know $r(C^*)$?

Binary search for r

$0 < r \le r_{max}$ **Guess:** $r = (0 + r_{max}) / 2$

⇒ Yes: K centers found with $2r$

or

No: r is too small

$r_0 < r \le r_1$

Do we really know $r(C^*)$?

Binary search for r

$0 < r \leq r_{max}$ **Guess:** $r = (0 + r_{max}) / 2$

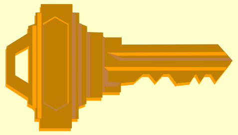
⇒ **Yes: K centers found with $2r$**

or

No: r is too small

$r_0 < r \leq r_1$ $r = (r_0 + r_1) / 2$

Do we really know $r(C^*)$?



Binary search for r

$0 < r \le r_{max}$ **Guess:** $r = (0 + r_{max}) / 2$

Yes: K centers found with $2r$

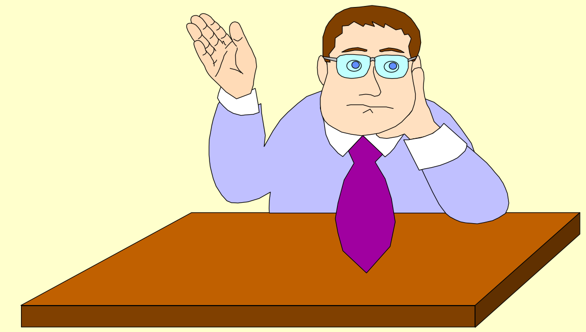
or

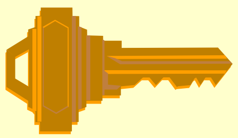
No: r is too small

$r_0 < r \le r_1$ $r = (r_0 + r_1) / 2$

Solution radius = $2r_1$ —— close to the true
2-approximation solution

Do we really know $r(C^*)$?



 **Binary search for r**

$0 < r \le r_{max}$ **Guess:** $r = (0 + r_{max}) / 2$

$$\begin{cases} \text{Yes: } K \text{ centers found with } 2r \\ \text{or} \\ \text{No: } r \text{ is too small} \end{cases}$$

$r_0 < r \le r_1$ $r = (r_0 + r_1) / 2$

Solution radius = $2r_1$ —— close to the true
2-approximation solution

We have a smarter solution without using the input r.

Center selection: greedy algorithm

Repeatedly choose next center to be site farthest from any existing center.

GREEDY-CENTER-SELECTION $(k, n, s_1, s_2, \ldots, s_n)$

$C \leftarrow \varnothing$.

REPEAT k times

  Select a site s_i with maximum distance $dist(s_i, C)$.

  $C \leftarrow C \cup s_i$.

RETURN C.

site farthest
from any center

Property. Upon termination, all centers in C are pairwise at least $r(C)$ apart.

Pf. By construction of algorithm.

Center selection: analysis of greedy algorithm

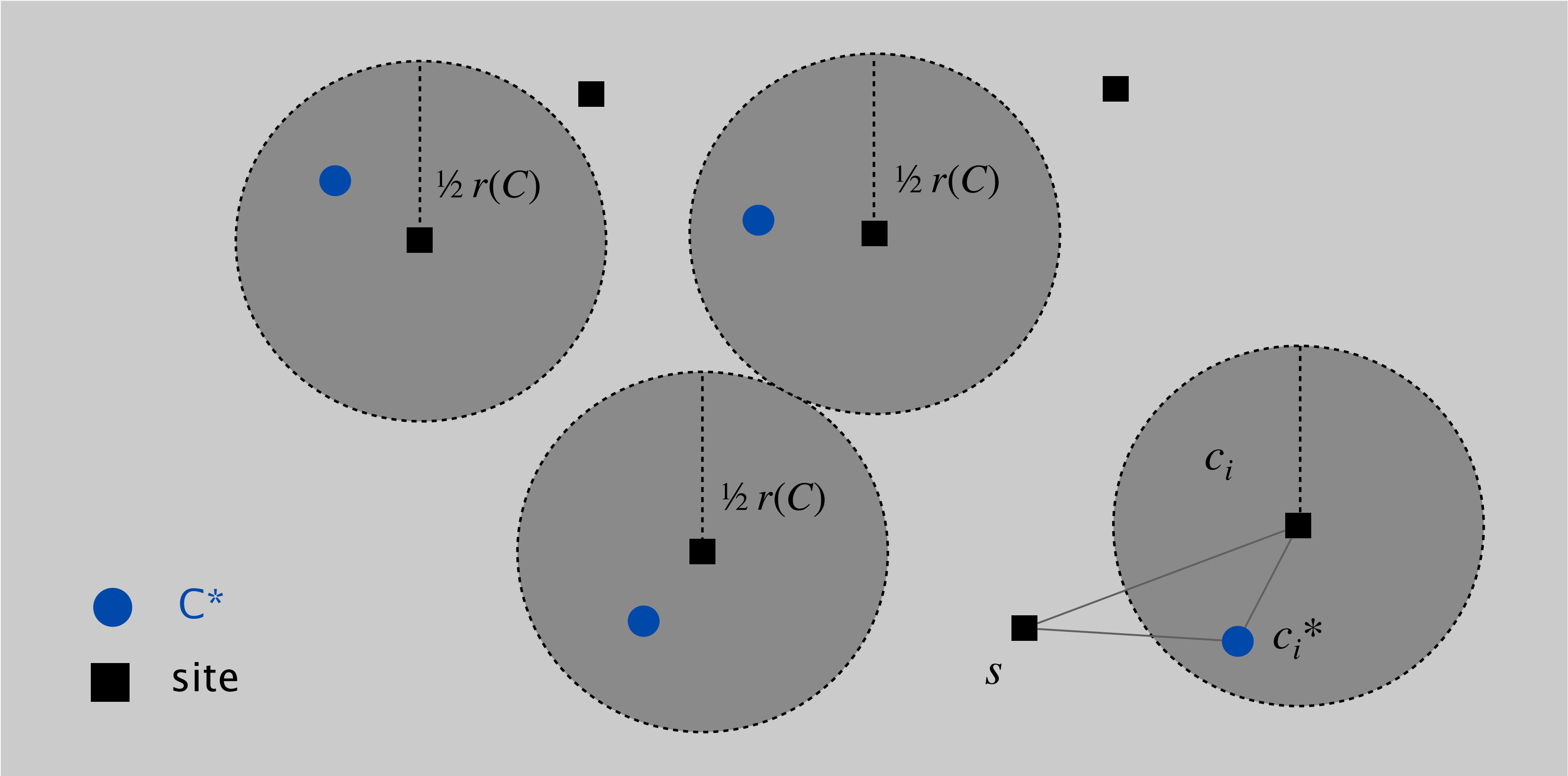
Lemma. Let C^* be an optimal set of centers. Then $r(C) \leq 2r(C^*)$.

Pf. [by contradiction] Assume $r(C^*) < \frac{1}{2}\, r(C)$.

- For each site $c_i \in C$, consider ball of radius $\frac{1}{2}\, r(C)$ around it.
- Exactly one c_i^* in each ball; let c_i be the site paired with c_i^*.
- Consider any site s and its closest center $c_i^* \in C^*$.
- $dist(s, C) \leq dist(s, c_i) \leq dist(s, c_i^*) + dist(c_i^*, c_i) \leq 2r(C^*)$.
- Thus, $r(C) \leq 2r(C^*)$. ▪

Δ-inequality

≤ $r(C^*)$ since c_i^* is closest center



Center selection

Lemma. Let C^* be an optimal set of centers. Then $r(C) \leq 2r(C^*)$.

Theorem. Greedy algorithm is a 2-approximation for center selection problem.

Remark. Greedy algorithm always places centers at sites, but is still within a factor of 2 of best solution that is allowed to place centers anywhere.

e.g., points in the plane

Question. Is there hope of a 3/2-approximation? 4/3?

Dominating set reduces to center selection

Theorem. Unless **P** = **NP**, there no ρ-approximation for center selection problem for any ρ < 2.

Pf. We show how we could use a $(2 - \varepsilon)$ approximation algorithm for CENTER-SELECTION selection to solve DOMINATING-SET in poly-time.

- Let $G = (V, E)$, k be an instance of DOMINATING-SET.
- Construct instance G' of CENTER-SELECTION with sites V and distances
  - $dist(u, v) = 1$ if $(u, v) \in E$
  - $dist(u, v) = 2$ if $(u, v) \notin E$
- Note that G' satisfies the triangle inequality.
- G has dominating set of size k iff there exists k centers C^* with $r(C^*) = 1$.
- Thus, if G has a dominating set of size k, a $(2 - \varepsilon)$-approximation algorithm for CENTER-SELECTION would find a solution C^* with $r(C^*) = 1$ since it cannot use any edge of distance 2. ▪

Outline: Approximation Algorithms

- Bin packing
- 0-1 knapsack
- K-center selection
- Take-home messages

Take-Home Messages

- Approximation algorithms: Deal with computational intractable (NP-hard) optimization problems, by relaxation on the optimality of solution to be find.

- Some definitions: approximation ratio, PTAS, and FPTAS.

- Two techniques in this lecture: Greedy search and rounding dynamic programming.

- Interesting in math properties: comparison on the unknown optimal solution, proving lower bounds.

Thanks for your attention!
Discussions?

Reference

Algorithm design: Chap. 11.

Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani: Sec. 9.2

Introduction to Algorithms (4th Edition): Chap. 35.

Data Structure and Algorithm Analysis in C (2nd Edition)： Sec 10.1.3.

https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/11ApproximationAlgorithms.pdf

Yitong Yin's lecture:

https://tcs.nju.edu.cn/wiki/index.php?
title=%E9%AB%98%E7%BA%A7%E7%AE%97%E6%B3%95_(Fall_2020)

https://tcs.nju.edu.cn/slides/aa2020/Greedy.pdf

https://tcs.nju.edu.cn/slides/aa2020/DP.pdf

Approximation Algorithms book: https://ics.uci.edu/~vazirani/book.pdf Chap. 8, 9.